Adobe

Adobe Flex 2

Flex 2 Developer's Guide

Flex™ 2 Developer's Guide

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Contents

PART 2: BUILDING USER INTERFACES FOR FLEX APPLICATIONS

PART 7: CHARTING COMPONENTS

CHAPTER 1

About Flex Documentation

1

Flex 2 Developer's Guide provides the tools for you to develop Internet applications by using Adobe® Flex™ 2. This book is intended for application developers who are learning Flex or want to extended their Flex programming knowledge. It provides a solid grounding in the tools that Flex provides to develop applications.

Contents

Using this manual

This manual can help anyone who is developing Flex applications. However, this manual is most useful if you have basic experience using Flex, or have read *Getting Started with Flex 2*. *Getting Started with Flex 2* provides an introduction to Flex and helps you develop the basic knowledge that makes using this manual easier.

Flex 2 Developer's Guide is divided into the following parts:

Part	Description
Part 1, "Using Flex Programming Languages"	Describes how to use MXML and ActionScript.
Part 2, "Building User Interfaces for Flex Applications"	Describes how to use Flex components to build the user interface to your application.
Part 3, "Customizing the User Interface"	Describes how to improve the user experience by adding additional functionality to your application.
Part 4, "Flex Programming Topics"	Describes some advanced programming techniques that you can use to make your applications more interactive and expressive.

Part	Description
Part 5, "Flex Data Features"	Describes how to use Flex data representation and data features.
Part 6, "Data Access and Interconnectivity"	Describes the Flex features that let you work with external data. Includes Adobe® Flex™ Data Services features.
Part 7, "Charting Components"	Describes how to use charting components.

Accessing the Flex documentation

The Flex documentation is designed to provide support for the complete spectrum of participants.

Documentation set

The Flex documentation set includes the following titles:

Book	Description
Flex 2 Developer's Guide	Describes how to develop your dynamic web applications.
Getting Started with Flex 2	Contains an overview of Flex features and application development procedures.
Building and Deploying Flex 2 Applications	Describes how to build and deploy Flex applications.
Creating and Extending Flex 2 Components	Describes how to create and extend Flex components.
Migrating Applications to Flex 2	Provides an overview of the migration process, as well as detailed descriptions of changes in Flex and ActionScript.
Using Flex Builder 2	Contains comprehensive information about all Adobe® Flex™ Builder™ 2 features, for every level of Flex Builder users.
Adobe Flex 2 Language Reference	Provides descriptions, syntax, usage, and code examples for the Flex API.

Viewing online documentation

All Flex documentation is available online in HTML and Adobe® Portable Document Format (PDF) files from the Adobe website. It is also available from the Adobe® Flex™ Builder™ Help menu.

Typographical conventions

The following typographical conventions are used in this book:

- *Italic font* indicates a value that should be replaced (for example, in a folder path).
- `Code font` indicates code.
- *`Code font italic`* indicates a parameter.
- **Boldface font** indicates a verbatim entry.

PART 1

Using Flex Programming Languages

1

This part describes how to use MXML and ActionScript, the Adobe Flex 2 programming languages.

The following topics are included:

CHAPTER 2

Developing Applications in MXML

2

MXML is an XML language that you use to lay out user-interface components for Adobe Flex applications. You also use MXML to declaratively define nonvisual aspects of an application, such as access to server-side data sources and data bindings between user-interface components and server-side data sources. This topic describes MXML and how you use MXML to develop Flex applications.

For information on MXML syntax, see Chapter 3, "MXML Syntax," on page 41.

Contents

About MXML

You use two languages to write Flex applications: MXML and ActionScript. MXML is an XML markup language that you use to lay out user-interface components. You also use MXML to declaratively define nonvisual aspects of an application, such as access to data sources on the server and data bindings between user-interface components and data sources on the server.

Like HTML, MXML provides tags that define user interfaces. MXML will seem very familiar if you have worked with HTML. However, MXML is more structured than HTML, and it provides a much richer tag set. For example, MXML includes tags for visual components such as data grids, trees, tab navigators, accordions, and menus, as well as nonvisual components that provide web service connections, data binding, and animation effects. You can also extend MXML with custom components that you reference as MXML tags.

One of the biggest differences between MXML and HTML is that MXML-defined applications are compiled into SWF files and rendered by Adobe® Flash® Player, which provides a richer and more dynamic user interface than page-based HTML applications provide.

You can write an MXML application in a single file or in multiple files. MXML also supports custom components written in MXML and ActionScript files.

Writing a simple application

Because MXML files are ordinary XML files, you have a wide choice of development environments. You can write MXML code in a simple text editor, a dedicated XML editor, or an integrated development environment (IDE) that supports text editing. Flex supplies a dedicated IDE, called Adobe Flex Builder, that you can use to develop your applications.

The following example shows a simple "Hello World" application that contains just an `<mx:Application>` tag and two child tags, the `<mx:Panel>` tag and the `<mx:Label>` tag. The `<mx:Application>` tag defines the Application container that is always the root tag of a Flex application. The `<mx:Panel>` tag defines a Panel container that includes a title bar, a title, a status message, a border, and a content area for its children. The `<mx:Label>` tag represents a Label control, a very simple user interface component that displays text.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <mx:Label text="Hello World!" fontWeight="bold" fontSize="24"/>

  </mx:Panel>
</mx:Application>
```

Save this code to a file named hello.mxml. MXML filenames must end in a lowercase .mxml file extension.

The following image shows the "Hello World" application rendered in a web browser window:

About XML encoding

The first line of the document specifies an optional declaration of the XML version. It is good practice to include encoding information that specifies how the MXML file is encoded. Many editors let you select from a range of file encoding options. On North American operating systems, ISO-8859-1 is the dominant encoding format, and most programs use that format by default. You can use the UTF-8 encoding format to ensure maximum platform compatibility. UTF-8 provides a unique number for every character in a file, and it is platform-, program-, and language-independent. If you specify an encoding format, it must match the file encoding you use. The following example shows an XML declaration tag that specifies the UTF-8 encoding format:

```
<?xml version="1.0" encoding="utf-8"?>
```

About the <mx:Application> tag

In addition to being the root tag of a Flex application, the `<mx:Application>` tag represents an Application container. A *container* is a user-interface component that contains other components and has built-in layout rules for positioning its child components. By default, an Application container lays out its children vertically from top to bottom. You can nest other types of containers inside an Application container, such as the Panel container shown above, to position user-interface components according to other rules. For more information, see Chapter 6, "Using Flex Visual Components," on page 133.

About MXML tag properties

The properties of an MXML tag, such as the `text`, `fontWeight`, and `fontSize` properties of the `<mx:Label>` tag, let you declaratively configure the initial state of the component. You can use ActionScript code in an `<mx:Script>` tag to change the state of a component at run time. For more information, see Chapter 4, "Using ActionScript," on page 55.

Compiling MXML to SWF Files

You can deploy your application as a compiled SWF file or, if you have Adobe Flex Data Services, you can deploy your application as a set of MXML and AS files.

If you are using Flex Builder, you compile and run the compiled SWF file from within Flex Builder. After your application executes correctly, you deploy it by copying it to a directory on your web server or application server. Users then access the deployed SWF file by making an HTTP request in the form:

```
http://hostname/path/filename.swf
```

The Flex also provides a command-line MXML compiler, mxmlc, that lets you compile MXML files. You can use mxmlc to compile hello.mxml from a command line, as the following example shows:

```
cd flexInstallDir/bin
mxmlc --show-actionscript-warnings=true --strict=true
  c:/appDir/hello.mxml
```

In this example, *flexInstallDir* is the Flex installation directory, and *appDir* is the directory containing hello.mxml. The resultant SWF file, hello.swf, is written to the same directory as hello.mxml.

For more information about mxmlc, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*. For more information about the debugger version of Flash Player, see Chapter 11, "Logging," in *Building and Deploying Flex 2 Applications*.

The relationship of MXML tags to ActionScript classes

Adobe implemented Flex as an ActionScript class library. That class library contains components (containers and controls), manager classes, data-service classes, and classes for all other features. You develop applications by using the MXML and ActionScript languages with the class library.

MXML tags correspond to ActionScript classes or properties of classes. Flex parses MXML tags and compiles a SWF file that contains the corresponding ActionScript objects. For example, Flex provides the ActionScript Button class that defines the Flex Button control. In MXML, you create a Button control by using the following MXML statement:

```
<mx:Button label="Submit"/>
```

When you declare a control using an MXML tag, you create an instance object of that class. This MXML statement creates a Button object, and initializes the label property of the Button object to the string Submit.

An MXML tag that corresponds to an ActionScript class uses the same naming conventions as the ActionScript class. Class names begin with an uppercase letter, and uppercase letters separate the words in class names. Every MXML tag attribute corresponds to a property of the ActionScript object, a style applied to the object, or an event listener for the object. For a complete description of the Flex class library and MXML tag syntax, see the *Adobe Flex 2 Language Reference*.

Understanding a Flex application structure

You can write an MXML application in a single file or in multiple files. You typically define a main file that contains the `<mx:Application>` tag. From within your main file, you can then reference additional files written in MXML, ActionScript, or a combination of the two languages.

A common coding practice is to divide your Flex application into functional units, or modules, where watch module performs a discrete task. In Flex, you can divide your application into separate MXML files and ActionScript files, where each file corresponds to a different module. By dividing your application into modules you provide many benefits, including the following:

Ease of development Different developers or development groups can develop and debug modules independently of each other.

Reusability You can reuse modules in different application so that you do not have to duplicate your work.

Maintainability You can isolate and debug errors faster than you could if you application was developed in a single file.

In Flex, a module corresponds to a custom component implement either in MXML or in ActionScript. These custom components can reference other custom components. There is no restriction on the level of nesting of component references in Flex. You define your components as required by your application.

Developing applications

MXML development is based on the same iterative process used for other types of web application files such as HTML, JavaServer Pages (JSP), Active Server Pages (ASP), and ColdFusion Markup Language (CFML). Developing a useful Flex application is as easy as opening your favorite text editor, typing some XML tags, saving the file, requesting the file's URL in a web browser, and then repeating the same process.

Flex also provides tools for code debugging; for more information, see Chapter 12, "Using the Command-Line Debugger," in *Building and Deploying Flex 2 Applications*.

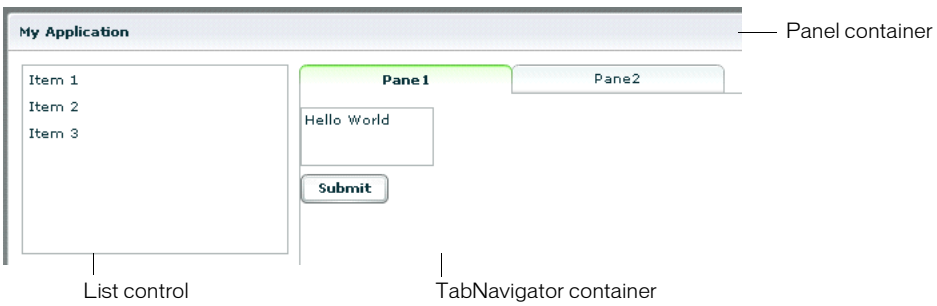
Laying out a user interface using containers

In the Flex model-view design pattern, user interface components represent the view. The MXML language supports two types of user interface components: controls and containers. Controls are form elements, such as buttons, text fields, and list boxes. Containers are rectangular regions of the screen that contain controls and other containers.

You use container components for laying out a user interface, and for controlling user navigation through the application. Examples of layout containers include the HBox container for laying out child components horizontally, the VBox container for laying out child components vertically, and the Grid container for laying out child components in rows and columns. Examples of navigator containers include the TabNavigator container for creating tabbed panels, the Accordion navigator container for creating collapsible panels, and the ViewStack navigator container for laying out panels on top of each other.

The Container class is the base class of all Flex container classes. Containers that extend the Container class add their own functionality for laying out child components. Typical properties of a container tag include `id`, `width`, and `height`. For more information about the standard Flex containers, see Chapter 13, "Introducing Containers," on page 491.

The following image shows an example Flex application that contains a List control on the left side of the user interface and a TabNavigator container on the right side. Both controls are enclosed in a Panel container:



Use the following code to implement this application:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10">

    <mx:HBox>
      <!-- List with three items -->
      <mx:List>
        <mx:dataProvider>
```

```
            <mx:Array>
              <mx:String>Item 1</mx:String>
              <mx:String>Item 2</mx:String>
              <mx:String>Item 3</mx:String>
            </mx:Array>
          </mx:dataProvider>
        </mx:List>

        <!-- First pane of TabNavigator -->
        <mx:TabNavigator borderStyle="solid">
          <mx:VBox label="Pane1" width="300" height="150">
            <mx:TextArea text="Hello World"/>
            <mx:Button label="Submit"/>
          </mx:VBox>

          <!-- Second pane of TabNavigator -->
          <mx:VBox label="Pane2" width="300" height="150">
            <!-- Stock view goes here -->
          </mx:VBox>

        </mx:TabNavigator>
      </mx:HBox>
    </mx:Panel>
</mx:Application>
```

The List control and TabNavigator container are laid out side by side because they are in an HBox container. The controls in the TabNavigator container are laid out from top to bottom because they are in a VBox container.

For more information about laying out user-interface components, see Chapter 6, "Using Flex Visual Components," on page 133.

Adding user interface controls

Flex includes a large selection of user interface components, such as Button, TextInput, and ComboBox controls. After you define the layout and navigation of your application by using container components, you add the user interface controls.

The following example contains an HBox (horizontal box) container with two child controls, a TextInput control and a Button control. An HBox container lays out its children horizontally.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:HBox>
    <mx:TextInput id="myText"/>
    <mx:Button click="storeZipInDatabase(myText.text)"/>
  </mx:HBox>
```

```
</mx:Application>
```

Typical properties of a control tag include `id`, `width`, `height`, `fontSize`, `color`, event listeners for events such as `click` and `change`, and effect triggers such as `showEffect` and `rollOverEffect`. For information about the standard Flex controls, see Chapter 9, "Using Controls," on page 259.

Using the id property with MXML tags

With a few exceptions (see "MXML tag rules" on page 54), an MXML tag has an optional `id` property, which must be unique within the MXML file. If a tag has an `id` property, you can reference the corresponding object in ActionScript.

In the following example, results from a web service request are traced in the `writeToLog` function:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
...

  <mx:VBox>
    <mx:TextInput id="myText" text="Hello World!" />
    <mx:Button id="mybutton" label="Get Weather" click="writeToLog();"/>
  </mx:VBox>
  <mx:Script>
    <![CDATA[
      private function writeToLog():void {
        trace(myText.text);
      }
    ]]>
  </mx:Script>
</mx:Application>
```

This code causes the MXML compiler to autogenerate a public variable named `myText` that contains a reference to that TextInput instance. This autogenerated variable lets you access the component instance in ActionScript. You can explicitly refer to the TextInput control's instance with its `id` instance reference in any ActionScript class or script block. By referring to a component's instance, you can modify its properties and call its methods.

Because each `id` value in an MXML file is unique, all objects in a file are part of the same flat namespace. You do not qualify an object by referencing its parent with dot notation, as in `myVBox.myText.text`.

For more information, see "Referring to Flex components" on page 60.

Using XML namespaces

In an XML document, tags are associated with a namespace. XML namespaces let you refer to more than one set of XML tags in the same XML document. The `xmlns` property in an MXML tag specifies an XML namespace. To use the default namespace, specify no prefix. To use additional tags, specify a tag prefix and a namespace. For example, the `xmlns` property in the following `<mx:Application>` tag indicates that tags in the MXML namespace use the prefix *mx:*. The Universal Resource Identifier (URI) for the MXML namespace is http://www.adobe.com/2006/mxml.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
```

XML namespaces give you the ability to use custom tags that are not in the MXML namespace. The following example shows an application that contains a custom tag called CustomBox. The namespace value `containers.boxes.*` indicates that an MXML component called CustomBox is in the containers/boxes directory.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:MyComps="containers.boxes.*">

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10">

    <MyComps:CustomBox/>

  </mx:Panel>
</mx:Application>
```

The containers/boxes directory can be a subdirectory of the directory that contains the application file, or it can be a subdirectory of one of the ActionScript source path directories assigned in the flex-config.xml file. If copies of the same file exist in both places, Flex uses the file in the application file directory. The prefix name is arbitrary, but it must be used as declared.

When using a component contained in a SWC file, the package name and the namespace must match, even though the SWC file is in the same directory as the MXML file that uses it. A SWC file is an archive file for Flex components. SWC files make it easy to exchange components among Flex developers. You need only exchange a single file, rather than the MXML or ActionScript files and images and other resource files. Also, the SWF file inside a SWC file is compiled, which means that the code is obfuscated from casual view. For more information on SWC files, see Chapter 9, “Using the Flex Compilers,” in *Building and Deploying Flex 2 Applications*.

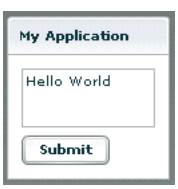
Using MXML to trigger run-time code

Flex applications are driven by run-time events, such as when a user selects a Button control. You can specify *event listeners*, which consist of code for handling run-time events, in the event properties of MXML tags. For example, the `<mx:Button>` tag has a `click` event property in which you can specify ActionScript code that executes when the Button control is clicked at run time. You can specify simple event listener code directly in event properties. To use more complex code, you can specify the name of an ActionScript function defined in an `<mx:Script>` tag.

The following example shows an application that contains a Button control and a TextArea control. The `click` property of the Button control contains a simple event listener that sets the value of the TextArea control's `text` property to the text *Hello World*.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10">

    <mx:TextArea id="textarea1"/>
    <mx:Button label="Submit" click="textarea1.text='Hello World';"/>

  </mx:Panel>
</mx:Application>
```

The following image shows the application rendered in a web browser window:



The following example shows the code for a version of the application in which the event listener is contained in an ActionScript function in an `<mx:Script>` tag:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      private function hello():void {
        textarea1.text="Hello World";
      }
    ]]>
  </mx:Script>
```

```
  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <mx:TextArea id="textarea1"/>
    <mx:Button label="Submit" click="hello();"/>

  </mx:Panel>
</mx:Application>
```

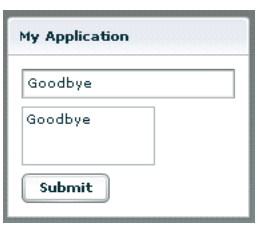
For more information about using ActionScript with MXML, see Chapter 4, "Using ActionScript," on page 55.

Binding data between components

Flex provides simple syntax for binding the properties of components to each other. In the following example, the value inside the curly braces ({ }) binds the `text` property of a TextArea control to the `text` property of a TextInput control. When the application initializes, both controls display the text *Hello*. When the user clicks the Button control, both controls display the text *Goodbye*.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <mx:TextInput id="textinput1" text="Hello"/>
    <mx:TextArea id="textarea1" text="{textinput1.text}"/>
    <mx:Button label="Submit" click="textinput1.text='Goodbye';"/>

  </mx:Panel>
</mx:Application>
```

The following image shows the application rendered in a web browser window after the user clicks the Submit button:



As an alternative to the curly braces ({ }) syntax, you can use the `<mx:Binding>` tag, in which you specify the source and destination of a binding. For more information about data binding, see Chapter 39, "Storing Data," on page 1269.

Using RPC services

Remote-procedure-call (RPC) services let your application interact with remote servers to provide data to your applications, or for your application to send data to a server.

Flex is designed to interact with several types of RPC services that provide access to local and remote server-side logic. For example, a Flex application can connect to a web service that uses the Simple Object Access Protocol (SOAP), a Java object residing on the same application server as Flex using AMF, or an HTTP URL that returns XML. AMF is the protocol used in Flash Remoting MX.

The MXML components that provide data access are called *RPC components*. MXML includes the following types of RPC components:

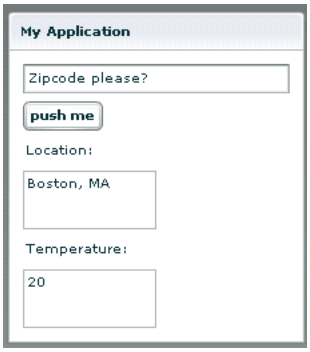
- **WebService** provides access to SOAP-based web services
- **HTTPService** provides access to HTTP URLs that return data
- **RemoteObject** provides access to Java objects using the AMF protocol (Flex Data Services only)

The following example shows an application that calls a web service that provides weather information, and displays the current temperature for a given ZIP code. The application binds the ZIP code that a user enters in a TextInput control to a web service input parameter. It binds the current temperature value contained in the web service result to a TextArea control.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <!-- Define the web service connection
    (the specified WSDL URL is not functional). -->
  <mx:WebService id="WeatherService"
    wsdl="http:/example.com/ws/WeatherService?wsdl"
    useProxy="false">

    <!-- Bind the value of the ZIP code entered in the TextInput control
      to the ZipCode parameter of the GetWeather operation. -->
    <mx:operation name="GetWeather">
      <mx:request>
        <ZipCode>{zip.text}</ZipCode>
      </mx:request>
    </mx:operation>
  </mx:WebService>

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <!-- Provide a ZIP code in a TextInput control. -->
    <mx:TextInput id="zip" width="200" text="Zipcode please?"/>
```

```
    <!-- Call the web service operation with a Button click. -->
    <mx:Button width="60" label="Get Weather"
      click="WeatherService.GetWeather.send();"/>

    <!-- Display the location for the specified ZIP code. -->
    <mx:Label text="Location:"/>
    <mx:TextArea text="{WeatherService.GetWeather.lastResult.Location}"/>

    <!-- Display the current temperature for the specified ZIP code. -->
    <mx:Label text="Temperature:"/>
    <mx:TextArea
      text="{WeatherService.GetWeather.lastResult.CurrentTemp}"/>
  </mx:Panel>
</mx:Application>
```

The following image shows the application rendered in a web browser window:



For more information about using RPC services, see Chapter 44, “Understanding RPC Components,” on page 1399.

Storing data in a data model

You can use a data model to store application-specific data. A *data model* is an ActionScript object that provides properties for storing data, and optionally contains methods for additional functionality. Data models provide a way to store data in the Flex application before it is sent to the server, or to store data sent from the server before using it in the application.

You can declare a simple data model that does not require methods in an <mx:Model>, <mx:XML>, or <mx:XMLList> tag. The following example shows an application that contains TextInput controls for entering personal contact information and a data model, represented by the <mx:Model> tag, for storing the contact information:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <!-- A data model called "contact" stores contact information.
    The text property of each TextInput control shown above
    is passed to a field of the data model. -->
  <mx:Model id="contact">
    <info>
      <homePhone>{homePhoneInput.text}</homePhone>
      <cellPhone>{cellPhoneInput.text}</cellPhone>
      <email>{emailInput.text}</email>
    </info>
  </mx:Model>

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >
    <!-- The user enters contact information in TextInput controls. -->
    <mx:TextInput id="homePhoneInput"
      text="This isn't a valid phone number."/>
    <mx:TextInput id="cellPhoneInput" text="(999)999-999"/>
    <mx:TextInput id="emailInput" text="me@somewhere.net"/>
  </mx:Panel>

</mx:Application>
```

Validating data

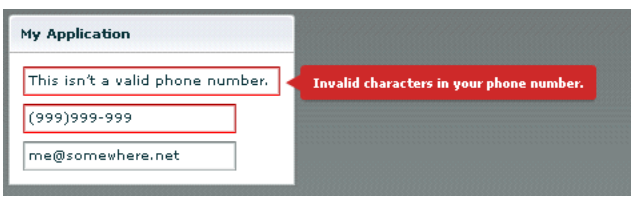
You can use validator components to validate data stored in a data model, or in a Flex user-interface component. Flex includes a set of standard validator components. You can also create your own.

The following example uses validator components for validating that the expected type of data is entered in the TextInput fields. Validation is triggered automatically when the users edits a TextInput control. If validation fails, the user receives immediate visual feedback.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <!-- A data model called "contact" stores contact information.
    The text property of each TextInput control shown above
    is passed to a field of the data model. -->
  <mx:Model id="contact">
    <info>
```

```
      <homePhone>{homePhoneInput.text}</homePhone>
      <cellPhone>{cellPhoneInput.text}</cellPhone>
      <email>{emailInput.text}</email>
    </info>
  </mx:Model>

  <!-- Validator components validate data entered into the TextInput
  controls. -->
  <mx:PhoneNumberValidator id="pnV"
    source="{homePhoneInput}" property="text"/>
  <mx:PhoneNumberValidator id="pnV2"
    source="{cellPhoneInput}" property="text"/>
  <mx:EmailValidator id="emV" source="{emailInput}" property="text" />

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >
    <!-- The user enters contact information in TextInput controls. -->
    <mx:TextInput id="homePhoneInput"
      text="This isn't a valid phone number."/>
    <mx:TextInput id="cellPhoneInput" text="(999)999-999"/>
    <mx:TextInput id="emailInput" text="me@somewhere.net"/>
  </mx:Panel>

</mx:Application>
```

The following image shows the application rendered in a web browser window:



For more information about using data models, see Chapter 39, "Storing Data," on page 1269. For more information on validators, see Chapter 40, "Validating Data," on page 1281.

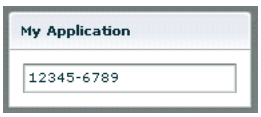
Formatting data

Formatter components are ActionScript components that perform a one-way conversion of raw data to a formatted string. They are triggered just before data is displayed in a text field. Flex includes a set of standard formatters. You can also create your own formatters. The following example shows an application that uses the standard ZipCodeFormatter component to format the value of a variable:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
```

```
  <!-- Declare a ZipCodeFormatter and define parameters. -->
  <mx:ZipCodeFormatter id="ZipCodeDisplay" formatString="#####-####" />

  <mx:Script>
    <![CDATA[
      private var storedZipCode:Number=123456789;
    ]]>
  </mx:Script>

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput text="{ZipCodeDisplay.format(storedZipCode)}" />

  </mx:Panel>
</mx:Application>
```

The following image shows the application rendered in a web browser window:



For more information about formatter components, see Chapter 41, “Formatting Data,” on page 1327.

Using Cascading Style Sheets (CSS)

You can use style sheets based on the Cascading Style Sheets (CSS) standard to declare styles to Flex components. The MXML `<mx:Style>` tag contains inline style definitions or a reference to an external file that contains style definitions.

The `<mx:Style>` tag must be an immediate child of the root tag of the MXML file. You can apply styles to an individual component using a class selector, or to all components of a certain type using a type selector.

The following example defines a class selector and a type selector in the `<mx:Style>` tag. Both the class selector and the type selector are applied to the Button control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
    .myClass { color: Red } /* class selector */
    Button { font-size: 18pt} /* type selector */
  </mx:Style>
```

```
  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <mx:Button styleName="myClass" label="This is red 18 point text."/>

  </mx:Panel>
</mx:Application>
```

A class selector in a style definition, defined as a label preceded by a period, defines a new named style, such as myClass in the preceding example. After you define it, you can apply the style to any component using the `styleName` property. In the preceding example, you apply the style to the Button control to set the font color to red.

A type selector applies a style to all instances of a particular component type. In the preceding example, you set the font size for all Button controls to 18 points.

The following image shows the application rendered in a web browser window:

For more information about using Cascading Style Sheets, see Chapter 18, “Using Styles and Themes,” on page 697.

Using skins

Skinning is the process of changing the appearance of a component by modifying or replacing its graphical elements. These graphical elements can be made up of images or the output of drawing API methods. They are known as *symbols*. You can reskin Flex components without changing their functionality. A file that contains new skins for use in your Flex applications is known as a *theme*.

There are two types of skins in Flex: graphical and programmatic. *Graphical skins* are Adobe Flash symbols that you can change directly in the Macromedia® Flash® Professional 8 from Adobe® authoring environment. You draw *programmatic skins* by using ActionScript statements and define these skins in class files. Sometimes it is more advantageous to reskin a component graphically, and in some cases it makes more sense to reskin a component programmatically. You cannot combine graphical and programmatic skins in a single theme file.

For more information about using skins, see Chapter 20, “Using Skins,” on page 805.

Using effects

An *effect* is a change to a component that occurs over a brief period of time. Examples of effects are fading, resizing, and moving a component. An effect is combined with a *trigger*, such as a mouse click on a component, a component getting focus, or a component becoming visible, to form a *behavior*. In MXML, you apply effects as properties of a control or container. Flex provides a set of built-in effects with default properties.

The following example shows an application that contains a Button control with its `rollOverEffect` property set to use the WipeLeft effect when the user moves the mouse over it:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <!-- Define the effect. -->
  <mx:WipeLeft id="myWL" duration="1000"/>

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <!-- Assign effect to targets. -->
    <mx:Button id="myButton" rollOverEffect="{myWL}"/>
  </mx:Panel>
</mx:Application>
```

For more information about effects, see Chapter 17, "Using Behaviors," on page 649.

Defining custom MXML components

Custom MXML components are MXML files that you create and use as custom MXML tags in other MXML files. They encapsulate and extend the functionality of existing Flex components. Just like MXML application files, MXML component files can contain a mix of MXML tags and ActionScript code. The name of the MXML file becomes the class name with which you refer to the component in another MXML file.

NOTE

You cannot access custom MXML component URLs directly in a web browser.

The following example shows a custom ComboBox control that is prepopulated with list items:

```
<?xml version="1.0"?>
<!-- MyComboBox.mxml -->

<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:ComboBox >
```

```
    <mx:dataProvider>
      <mx:String>Dogs</mx:String>
      <mx:String>Cats</mx:String>
      <mx:String>Mice</mx:String>
    </mx:dataProvider>
  </mx:ComboBox>

</mx:VBox>
```

The following example shows an application that uses the MyComboBox component as a custom tag. The value * assigns the *local* namespace to the current directory.

```
<?xml version="1.0"?>
<!-- MyApplication.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:MyComps="containers.boxes.*">

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
    paddingLeft="10" paddingRight="10" >

    <MyComps:MyComboBox/>

  </mx:Panel>
</mx:Application>
```

The following image shows the application rendered in a web browser window:

For more information about MXML components, see Chapter 7, “Creating Simple MXML Components,” in *Creating and Extending Flex 2 Components*.

You can also define custom Flex components in ActionScript. For more information, see Chapter 9, “Creating Simple Visual Components in ActionScript,” in *Creating and Extending Flex 2 Components*.

CHAPTER 3

MXML Syntax

3

MXML is an XML language that you use to lay out user-interface components for Adobe Flex applications. This topic describes basic MXML syntax.

Contents

Basic MXML syntax

Most MXML tags correspond to ActionScript 3.0 classes or properties of classes. Flex parses MXML tags and compiles a SWF file that contains the corresponding ActionScript objects.

ActionScript 3.0 uses syntax based on the ECMAScript edition 4 draft language specification. ActionScript 3.0 includes the following features:

- Formal class definition syntax
- Formal packages structure
- Typing of variables, parameters, and return values (compile-time only)
- Implicit getters and setters that use the `get` and `set` keywords
- Inheritance
- Public and private members
- Static members
- Cast operator

For more information about ActionScript 3.0, see Chapter 7, “Using ActionScript,” on page 91.

Naming MXML files

MXML filenames must adhere to the following naming conventions:

- Filenames must be valid ActionScript identifiers, which means they must start with a letter or underscore character (_), and they can only contain letters and numbers and underscore characters after that.
- Filenames must not be the same as ActionScript class names, component `id` values, or the word *application*. Do not use filenames that match the names of MXML tags that are in the mx namespace.
- Filenames must end with a lowercase .mxml file extension.

Using tags that represent ActionScript classes

An MXML tag that corresponds to an ActionScript class uses the same naming conventions as the ActionScript class. Class names begin with a capital letter, and capital letters separate the words in class names. For example, when a tag corresponds to an ActionScript class, its properties correspond to the properties and events of that class.

Setting component properties

In MXML, a component property uses the same naming conventions as the corresponding ActionScript property. A property names begins with a lowercase letter, and capital letters separate words in the property names.

You can set most component properties as tag attributes, in the form:

```
<mx:Label width="50" height="25" text="Hello World"/>
```

You can set all component properties as child tags, in the form:

```
<mx:Label>
  <mx:width>50</mx:width>
  <mx:height>25</mx:height>
  <mx:text>Hello World</mx:text>
</mx:Label>
```

You often use child tags when setting the value of a property to a complex Object because it is not possible to specify a complex Object as the value of tag attribute. In the following example, you use child tags to set the data provider of a ComboBox control of an ArrayCollection object:

```
<mx:ComboBox>
  <mx:dataProvider>
    <mx:ArrayCollection>
      <mx:String>AK</mx:String>
```

```
      <mx:String>AL</mx:String>
      <mx:String>AR</mx:String>
    </mx:ArrayCollection>
  <mx:dataProvider>
</mx:ComboBox>
```

The one restriction on setting properties that use child tags is that the namespace prefix of a child tag, `mx:` in the previous example, must match the namespace prefix of the component tag.

Each of a component's properties is one of the following types:

- Scalar properties, such as a number or string
- Array of scalar values, such as an array of numbers or strings
- ActionScript object
- Array of ActionScript objects
- ActionScript properties
- XML data

Adobe recommends that you assign scalar values using tag attributes, and that you assign complex types, such as ActionScript objects, by using child tags.

Setting scalar properties

You usually specify the value of a scalar property as a property of a component tag, as the following example shows:

```
<mx:Label width="50" height="25" text="Hello World"/>
```

Setting properties using constants

The valid values of many component properties are defined by static constants, where these static constants are defined in an ActionScript class. In MXML, you can either use the static constant to set the property value, or use the value of the static constant, as the following example shows:

```
<!-- Set the property using the static constant. -->
<mx:HBox width="200" horizontalScrollPolicy="{ScrollPolicy.OFF}">
  ...
</mx:HBox>

<!-- Set the property using the value of the static constant. -->
<mx:HBox width="200" horizontalScrollPolicy="off">
  ...
</mx:HBox>
```

The HBox container defines a property named `horizontalScrollPolicy` which defines the operation of the container's horizontal scroll bar. In this example, you explicitly set the `horizontalScrollPolicy` property to disable the horizontal scroll bar.

In the first example, you set the `horizontalScrollPolicy` property using a static constant named `OFF`, which is defined in the ScrollPolicy class. In MXML, you must use data binding syntax when setting a property value to a static constant. The advantage of using the static constant is that the Flex compiler recognizes incorrect property values, and issues an error message at compile time.

Alternatively, you can set the value of the `horizontalScrollPolicy` property to the value of the static constant. The value of the `OFF` static constant is `"off"`. When you use the value of the static constant to set the property value, the Flex compiler cannot determine if you used an unsupported value. If you incorrectly set the property, you will not know until you get a run-time error.

In ActionScript, you should always use static constants to set property values, as the following example shows:

```
var myHBox:HBox = new HBox();
myHBox.horizontalScrollPolicy=ScrollPolicy.OFF;
```

Setting the default property

Many Flex components define a single default property. The *default property* is the MXML tag property that is implicit for content inside of the MXML tag if you do not explicitly specify a property. For example, consider the following MXML tag definition:

```
<mx:SomeTag>
  anything here
</mx:SomeTag>
```

If this tag defines a default property named `default_property`, the preceding tag definition is equivalent to the following code:

```
<mx:SomeTag>
  <default_property>
    anything here
  </default_property>
</mx:SomeTag>
```

It is also equivalent to the following code:

```
<mx:SomeTag default_property="anything here"/>
```

The default property provides a shorthand mechanism for setting a single property. For a ComboBox, the default property is the `dataProvider` property. Therefore the two ComboBox definitions in the following code are equivalent:

```
<?xml version="1.0"?>
<!-- mxml\DefProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <!-- Omit the default property. -->
    <mx:ComboBox>
        <mx:ArrayCollection>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
            <mx:String>AR</mx:String>
        </mx:ArrayCollection>
    </mx:ComboBox>

    <!-- Explicitly speficy the default property. -->
    <mx:ComboBox>
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:String>AK</mx:String>
                <mx:String>AL</mx:String>
                <mx:String>AR</mx:String>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:ComboBox>
</mx:Application>
```

Not all Flex components define a default property. See *Adobe Flex 2 Language Reference* for each component to determine its default property.

You can also define a default property when you create a custom component. For more information, see Chapter 5, "Using Metadata Tags in Custom Components," in *Creating and Extending Flex 2 Components*.

Escaping characters using the backslash character

When setting a property value in MXML, you can escape a reserved character by prefixing it with the backslash character, "\", as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml\EscapeChar.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Label text="\{\}"/>

</mx:Application>
```

In this example, you want to use literal curly brace characters ({ }) in a text string. But Flex uses curly braces to indicate a data binding operation. Therefore, you prefix each curly brace with the backslash character to cause the MXML compiler to interpret them as literal characters.

Setting String properties using the backslash character

The MXML compiler automatically escapes the backslash character in MXML when the character is part of the value specified to a property of type String. Therefore, it always converts `"\"` to `"\\"`.

This is necessary because the ActionScript compiler recognizes `"\\"` as the character sequence for a literal "\" character, and strips out the leading backslash when it initializes the property value.

NOTE Do not use the backslash character (\) as a separator in the path to an application asset. You should always use a forward slash character (/) as the separator.

Including a newline character in a String value

For properties of type String, you can insert a newline character in the String in two ways:

- By inserting the `&#13;` code in your String value in MXML
- By inserting `"\n"` in an ActionScript String variable used to initialize the MXML property

To use the `&#13;` code to insert a newline character, include that code in the property value in MXML, as the following example shows:

```
<mx:TextArea width="100%" text="Display&#13;Content"/>
```

To use an ActionScript String variable to insert a newline character, create an ActionScript variable, and then use data binding to set the property in MXML, as the following example shows:

```
<mx:Script>
  <![CDATA[
    [Bindable]
    public var myText:String = "Display" + "\n" + "Content";
  ]]>
</mx:Script>

<mx:TextArea width="100%" text="{myText}"/>
```

In this example, you set the `text` property of the TextArea control to a value that includes a newline character.

Notice that this example includes the [Bindable] metadata tag before the property definition. This metadata tag specifies that the myText property can be used as the source of a data binding expression. Data binding automatically copies the value of a source property of one object to the destination property of another object at run time when the source property changes.

If you omit this metadata tag, the compiler issues a warning message specifying that the property cannot be used as the source for data binding. For more information, see Chapter 38, "Binding Data," on page 1245.

Setting Arrays of scalar values

When a class has a property that takes an Array as its value, you can represent the property in MXML using child tags. The component in the following example has a dataProvider property that contains an Array of numbers:

```
<mx:List width="150">
  <mx:dataProvider>
    <mx:Array>
      <mx:Number>94062</mx:Number>
      <mx:Number>14850</mx:Number>
      <mx:Number>53402</mx:Number>
    </mx:Array>
  </mx:dataProvider>
</mx:List>
```

The <mx:Array> and </mx:Array> tags around the Array elements are optional. Therefore, you can also write this example as the following example shows:

```
<mx:List width="150">
  <mx:dataProvider>
    <mx:Number>94062</mx:Number>
    <mx:Number>14850</mx:Number>
    <mx:Number>53402</mx:Number>
  </mx:dataProvider>
</mx:List>
```

In this example, since the data type of the dataProvider property is defined as Array, Flex automatically converts the three number definitions into a three-element array.

Component developers may have specified additional information within the component definition that defines the data type of the Array elements. For example, if the developer specified that the dataProvider property only supports String elements, then this example would cause a compiler error because you specified numbers to it. The *Adobe Flex 2 Language Reference* documents the Array properties that define a required data type for the Array elements.

Setting Object properties

When a component has a property that takes an object as its value, you can represent the property in MXML using a child tag with tag attributes, as the following example shows:

```
<mynamespace:MyComponent>
  <mynamespace:nameOfProperty>
    <mynamespace:typeOfObject prop1="val1" prop2="val2"/>
  </mynamespace:nameOfProperty>
</mynamespace:MyComponent>
```

The following example shows an ActionScript class that defines an Address object. This object is used as a property of the PurchaseOrder component in the next example.

```
class Address
{
  public var name:String;
  public var street:String;
  public var city:String;
  public var state:String;
  public var zip:Number;
}
```

The following example shows an ActionScript class that defines a PurchaseOrder component that has a property type of Address:

```
import example.Address;

class PurchaseOrder {
  public var shippingAddress:Address;
  public var quantity:Number;
  ...
}
```

In MXML, you define the PurchaseOrder component as the following example shows:

```
<mynamespace:PurchaseOrder quantity="3" xmlns:e="example">
  <mynamespace:shippingAddress>
    <mynamespace:Address name="Fred" street="123 Elm St."/>
  </mynamespace:shippingAddress>
</mynamespace:PurchaseOrder>
```

If the value of the `shippingAddress` property is a subclass of Address (such as DomesticAddress), you can declare the property value, as the following example shows:

```
<mynamespace:PurchaseOrder quantity="3" xmlns:e="example">
  <mynamespace:shippingAddress>
    <mynamespace:DomesticAddress name="Fred" street="123 Elm St."/>
  </mynamespace:shippingAddress>
</mynamespace:PurchaseOrder>
```

If the property is explicitly typed as `Object` like the `value` property in the following example, you can specify an anonymous object using the `<mx:Object>` tag.

```
class ObjectHolder {
  public var value:Object
}
```

The following example shows how you specify an anonymous object as the value of the `value` property:

```
<mynamespace:ObjectHolder>
  <mynamespace:value>
    <mx:Object foo='bar' />
  </mynamespace:value>
</mynamespace:ObjectHolder>
```

Populating an Object with an Array

When a component has a property of type Object that takes an Array as its value, you can represent the property in MXML using child tags, as the following example shows:

```
<mynamespace:MyComponent>
  <mynamespace:nameOfObjectProperty>
    <mx:Array>
      <mx:Number>94062</mx:Number>
      <mx:Number>14850</mx:Number>
      <mx:Number>53402</mx:Number>
    </mx:Array>
  </mynamespace:nameOfObjectProperty>
</mynamespace:MyComponent>
```

In this example, you initialize the Object to a three element array of numbers.

As described in the section "Setting Arrays of scalar values" on page 47, the `<mx:Array>` tag and the `</mx:Array>` tag around the Array elements are optional and may be omitted, as the following example shows:

```
<mynamespace:MyComponent>
  <mynamespace:nameOfObjectProperty>
    <mx:Number>94062</mx:Number>
    <mx:Number>14850</mx:Number>
    <mx:Number>53402</mx:Number>
  </mynamespace:nameOfObjectProperty>
</mynamespace:MyComponent>
```

The only exception to this rule is when you specify a single Array element for the Object property. In that case, Flex does not create an Object containing a single-element array, but instead creates an object and sets it to the specified value. This is a difference between the following:

```
object=[element] // Object containing a one-element array
object=element // object equals value
```

If you want to create a single element array, include the `<mx:Array>` and `</mx:Array>` tags around the array element, as the following example shows:

```
<mynamespace:MyComponent>
  <mynamespace:nameOfObjectProperty>
    <mx:Array>
      <mx:Number>94062</mx:Number>
    </mx:Array>
  </mynamespace:nameOfObjectProperty>
</mynamespace:MyComponent>
```

Populating Arrays of objects

When a component has a property that takes an Array of objects as its value, you can represent the property in MXML using child tags, as the following example shows:

```
<mynamespace:MyComponent>
  <mynamespace:nameOfProperty>
    <mx:Array>
      <mynamespace:objectType prop1="val1" prop2="val2"/>
      <mynamespace:objectType prop1="val1" prop2="val2"/>
      <mynamespace:objectType prop1="val1" prop2="val2"/>
    </mx:Array>
  </mynamespace:nameOfProperty>
</mynamespace:MyComponent>
```

The component in the following example contains an Array of ListItem objects. Each ListItem object has properties named `label` and `data`.

```
<mynamespace:MyComponent>
  <mynamespace:dataProvider>
    <mx:Array>
      <mynamespace:ListItem label="One" data="1"/>
      <mynamespace:ListItem label="Two" data="2"/>
    </mx:Array>
  </mynamespace:dataProvider>
</mynamespace:MyComponent>
```

The following example shows how you specify an anonymous object as the value of the `dataProvider` property:

```
<mynamespace:MyComponent>
  <mynamespace:dataProvider>
    <mx:Array>
      <mx:Object label="One" data="1"/>
      <mx:Object label="Two" data="2"/>
    </mx:Array>
  </mynamespace:dataProvider>
</mynamespace:MyComponent>
```

As described in the section "Setting Arrays of scalar values" on page 47, the `<mx:Array>` tag and the `</mx:Array>` tag around the Array elements are optional and may be omitted, as the following example shows:

```
<mynamespace:MyComponent>
  <mynamespace:dataProvider>
    <mx:Object label="One" data="1"/>
    <mx:Object label="Two" data="2"/>
  </mynamespace:dataProvider>
</mynamespace:MyComponent>
```

Setting properties that contain XML data

If a component contains a property that takes XML data, the value of the property is an XML fragment to which you can apply a namespace. In the following example, the `value` property of the MyComponent object is XML data:

```
<mynamespace:MyComponent>
  <mynamespace:value xmlns:a="http://www.example.com/myschema">
    <mx:XML>
      <a:purchaseorder>
        <a:billingaddress>
        ...
        </a:billingaddress>
        ...
    </a:purchaseorder>
  </mx:XML>
  </mynamespace:value>
</mynamespace:MyComponent>
```

Setting style and effect properties in MXML

A style or effect property of an MXML tag differs from other properties because it corresponds to an ActionScript style or effect, rather than to a property of an ActionScript class. You set these properties in ActionScript using the `setStyle(stylename, value)` method rather than `object.property=value` notation.

You define style or effect properties in ActionScript classes using the `[Style]` or `[Effect]` metadata tags, rather than defining them as ActionScript variables or setter/getter methods. For more information, see Chapter 5, "Using Metadata Tags in Custom Components," in *Creating and Extending Flex 2 Components*.

For example, you can set the `fontFamily` style property in MXML, as the following code shows:

```
<mx:TextArea id="myText" text="hello world" fontFamily="Tahoma"/>
```

This MXML code is equivalent to the following ActionScript code:

```
myText.setStyle("fontFamily", "Tahoma");
```

Setting event properties in MXML

An event property of an MXML tag lets you specify the event listener function for an event. This property correspond to setting the event listener in ActionScript using the `addEventListener()` method.

You define event properties in ActionScript classes using the `[Event]` metadata tags, rather than defining them as ActionScript variables or setter/getter methods. For more information, see Chapter 5, "Using Metadata Tags in Custom Components," in *Creating and Extending Flex 2 Components*.

For example, you can set the `creationComplete` event property in MXML, as the following code shows:

```
<mx:TextArea id="myText" creationComplete="creationCompleteHandler()"/>
```

This MXML code is equivalent to the following ActionScript code:

```
myText.addEventListener("creationComplete", creationCompleteHandler);
```

Specifying a URL value

Some MXML tags, such as the `<mx:Script>` tag, have a property that takes a URL of an external file as a value. For example, you can use the `source` property in an `<mx:Script>` tag to reference an external ActionScript file instead of typing ActionScript directly in the body of the `<mx:Script>` tag.

NOTE

You specify a script in the `source` property of an `<mx:Script>` tag. You do not specify ActionScript classes in the `source` property. For information on using ActionScript classes, see "Creating ActionScript components" on page 75 in the *Flex 2 Developer's Guide*.

MXML supports the following types of URLs:

- Absolute; for example:

  ```
  <mx:Style source="http://www.somesite.com/mystyles.css">
  ```

- A path used at run time that is relative to the context root of the Java web application in which a Flex application is running; for example:

  ```
  <mx:HTTPService url="@ContextRoot()/directory/myfile.xml"/>
  ```

- A path used at compile-time that is relative to the context root of the Java web application in which a Flex application is running; for example:

  ```
  <mx:Script source="/myscript.as"/>
  ```

- Relative to the current file location; for example:

  ```
  <mx:Script source="../myscript.as"/>
  ```

Specifying a RegExp value

For a property of type RegExp, you can specify its value in MXML using the following format:

"*/pattern/flags*"

pattern Specifies the regular expression within the two slashes. Both slashes are required.

flags (Optional) specifies any flags for the regular expression.

For example, the `regExpression` property of an MXML component is of type RegExp. Therefore, you can set its value, as the following example shows:

```
<mynamespace:MyComponent regExpression="/\Wcat/gi"/>
```

Or set it using child tags, as the following example shows:

```
<mynamespace:MyComponent>
  <mynamespace:regExpression>/\Wcat/gi</mynamespace:regExpression>
</mynamespace:MyComponent>
```

The *flags* portion of the regular expression is optional, so you can also specify it as the following example shows:

```
<mynamespace:MyComponent regExpression="/\Wcat/"/>
```

Using compiler tags

Compiler tags are tags that do not directly correspond to ActionScript objects or properties. The names of the following compiler tags have just the first letter capitalized:

- `<mx:Binding>`
- `<mx:Component>`
- `<mx:Metadata>`
- `<mx:Model>`
- `<mx:Script>`
- `<mx:Style>`
- `<mx:XML>`
- `<mx:XMLList>`

The following compiler tags are in all lowercase letters:

- `<mx:operation>`
- `<mx:request>`
- `<mx:method>`
- `<mx:arguments>`

MXML tag rules

MXML has the following syntax requirements:

- The `id` property is not required on any tag.
- The `id` property is not allowed on the root tag.
- Boolean properties take only `true` and `false` values.
- The `<mx:Binding>` tag requires both `source` and `destination` properties.
- The `<mx:Binding>` tag cannot contain an `id` property.
- The `<mx:WebService>` tag requires a `wsdl` value or `serviceName` value, and does not allow both.
- The `<mx:RemoteObject>` tag requires a `source` value or a `named` value, and does not allow both.
- The `<mx:HTTPService>` tag requires a `url` value or a `serviceName` value, and does not allow both.
- The `<mx:operation>` tag requires a `name` value, and does not allow duplicate `name` entries.
- The `<mx:operation>` tag cannot contain an `id` property.
- The `<mx:method>` tag requires a `name` value and does not allow duplicate `name` entries.
- The `<mx:method>` tag cannot contain an `id` property.

CHAPTER 4

Using ActionScript

4

Flex developers can use ActionScript to extend the functionality of their Adobe Flex applications. ActionScript provides flow control and object manipulation features that are not available in MXML. This topic explains how to use ActionScript in an MXML application. For a complete introduction to ActionScript and a reference for using the language, see *Programming ActionScript 3.0* and *ActionScript 3.0 Language Reference*.

Contents

Using ActionScript in Flex applications

Flex developers can use ActionScript to implement custom behavior within their Flex applications. You first use MXML tags to declare things like the containers, controls, effects, formatters, validators, and web services that your application requires, and to lay out its user interface. Each of these components provides the standard behavior you'd expect. For example, a button automatically highlights when you roll over it, without requiring you to write any ActionScript. But a declarative language like MXML is not appropriate for coding what you want to happen when the user clicks a button. For that, you need to use a procedural language like ActionScript, which offers executable methods, various types of storage variables, and flow control such as conditionals and loops. In a general sense, MXML implements the static aspects of your application, and ActionScript implements its dynamic aspects.

ActionScript is an object-oriented procedural programming language, based on the ECMAScript (ECMA-262) edition 4 draft language specification. You can use a variety of methods to mix ActionScript and MXML, including the following:

- Use ActionScript to define event listeners inside MXML event attributes.
- Add script blocks using the `<mx:Script>` tag.
- Include external ActionScript files.
- Import ActionScript classes.
- Create ActionScript components.

ActionScript compilation

Although a simple Flex application can be written in a single MXML or ActionScript (AS) file, most applications will be broken into multiple files. For example, it is common to move the `<mx:Script>` and `<mx:Style>` blocks for an `<mx:Application>` into separate AS and CSS files which the application then includes. It is also common for an application to import custom MXML and ActionScript components. These must be defined in other files, and MXML components may put their own `<mx:Script>` blocks into yet more AS files that they include. Components may also be imported from precompiled SWC files rather than source code. Finally, SWF files containing executable code can also be embedded in an application. The end result of all these input files is a single SWF file.

You can use ActionScript code fragments in a number of places within your MXML files. The Flex compiler transforms the main MXML file and other files it includes into a single ActionScript class. So, you cannot define classes or use statements outside of functions in MXML files and included ActionScript files.

You can reference imported ActionScript classes from your MXML application files, and those classes are added to the final SWF file. When the transformation to an ActionScript file is complete, Flex links all the ActionScript components and includes those classes in the final SWF file.

About generated ActionScript

When you write an MXML file and compile it, the Flex compiler creates a new class and generates ActionScript that the class uses. The following list describes the ways that MXML tags and ActionScript are used by the resulting class. You do not necessarily have to understand the information in this section to use Flex, but it can be useful for understanding what is happening out of view of the Flex developer.

An MXML application (a file starting with the `<mx:Application>` tag) defines a subclass of the Application class. Similarly, an MXML component (a file starting with some other component's tag, such as `<mx:Button>`) defines a subclass of that component.

The name of the subclass is the name of the file. The base class is the class of the top-level tag. An MXML application actually defines the following:

```
class MyApp extends Application
```

If MyButton.mxml starts with `<mx:Button>`, you are actually defining the following:

```
class MyButton extends Button
```

The variable and function declarations in an `<mx:Script>` block define properties and methods of the subclass.

Setting an `id` property on a component instance within a class results in a public variable being autogenerated in the subclass that contains a reference to that component instance. For example, if the `<mx:Button id="myButton"/>` tag is nested deeply inside several containers, you can still refer to it as `myButton`.

Event attributes become the bodies of autogenerated event listener methods in the subclass. For example:

```
<mx:Button id="myButton" click="foo = 1; doSomething()">
```

becomes

```
private function __myButton_click(event:MouseEvent):void {
  foo = 1;
  doSomething()
}
```

The event attributes become method bodies, so they can access the other properties and methods of the subclass.

All the ActionScript anywhere in an MXML file, whether in its `<mx:Script>` block or inside tags, executes with the `this` keyword referring to an instance of the subclass.

The public properties and methods of the class are accessible by ActionScript code in other components, as long as that code "dots down" (for example, `myCheckoutAccordion.myAddressForm.firstNameTextInput.text`) or reaches up using `parentDocument`, `parentApplication`, or `Application.application` to specify which component the property or method exists on.

Using ActionScript in MXML event handlers

One way to use ActionScript code in a Flex application is to include it within the MXML tag's event handler, as the following example shows:

```
<?xml version="1.0"?>
<!-- usingas/HelloWorldAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Panel title="My Application" >
     <mx:TextArea id="textarea1"/>
     <mx:Button label="Submit" click="textarea1.text='Hello World';"/>
  </mx:Panel>
</mx:Application>
```

In this example, you include ActionScript code for the body of the click event handler of the Button control. The MXML compiler takes the attribute `click="..."` and generates the following event handler method:

```
public function __myButton_click(event:MouseEvent):void {
  textarea1.text='Hello World';
}
```

When the user clicks the button, this code sets the value of the TextArea control's `text` property to the String "`Hello World`." In most cases, you do not need to look at the generated code, but it is useful to understand what happens when you write an inline event handler.

To see the generated code, set the value of the `keep-generated-actionscript` compiler option to `true`. The compiler then stores the *.as helper file in the /generated directory, which is a subdirectory of the location of the SWF file.

For more information about events, see Chapter 5, "Using Events," on page 83. For more information on using the command-line compilers, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Using ActionScript blocks in MXML files

You use the `<mx:Script>` tag to insert an ActionScript block in an MXML file. ActionScript blocks can contain ActionScript functions and variable declarations used in MXML applications. Code inside `<mx:Script>` tags can also declare constants (with the `const` statement) and namespaces (with `namespace`), include ActionScript files (with `include`), import declarations (with `import`), and use namespaces (with `use namespace`).

The `<mx:Script>` tag must be a child of the `<mx:Application>` or other top-level component tag.

Statements and expressions are allowed only if they are wrapped in a function. In addition, you cannot define new classes or interfaces in <mx:Script> blocks. Instead, you must place new classes or interfaces in separate AS files and import them.

All ActionScript in the block is added to the enclosing file's class when Flex compiles the application. The following example declares a variable and sets the value of that variable inside a function:

```
<?xml version="1.0"?>
<!-- usingas/StatementSyntax.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="doSomething()">
    <mx:Script><![CDATA[
            public var s:Boolean;
            public function doSomething():void {
                // The following statements must be inside a function.
                s = label1.visible;
                label1.text = String(s);
            }
    ]]></mx:Script>

    <mx:Label id="label1"/>

</mx:Application>
```

Most statements must be inside functions in an <mx:Script> block. However, the following statements can be outside functions:

- import
- var
- include
- const
- namespace
- use namespace

When using an <mx:Script> block, you should wrap the contents in a CDATA construct. This prevents the compiler from interpreting the contents of the script block as XML, and allows the ActionScript to be properly generated. Adobe recommends that you write all your <mx:Script> open and close tags as the following example shows:

```
<mx:Script>
  <![CDATA[
    ...
  ]]>
</mx:Script>
```

Flex does not parse text in a CDATA construct so that you can use XML-parsed characters such as angle brackets (< and >) and ampersand (&). For example, the following script that includes a less than (<) comparison must be in a CDATA construct:

```
<?xml version="1.0"?>
<!-- usingas/UsingCDATA.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="doSomething()">
    <mx:Script><![CDATA[
        public var m:Number;
        public var n:Number;
        public function doSomething():void {
            n = 42;
            m = 40;
            label1.text = String(n > m);
        }
    ]]></mx:Script>

    <mx:Label id="label1"/>

</mx:Application>
```

Accessing ActionScript documentation

The ActionScript 3.0 programming language can be used from within several development environments, including Flash Professional and Adobe Flex Builder.

The Flex documentation includes *Programming ActionScript 3.0*, which describes the ActionScript language. The ActionScript API reference is included as part of the *Adobe Flex 2 Language Reference*.

Working with Flex components

The primary use of ActionScript in your Flex applications is probably going to be for working with the visual controls and containers in your application. This section describes techniques for doing this, including how to reference a Flex control in ActionScript and how to manipulate properties during the control's and container's instantiation.

Referring to Flex components

To work with a component in ActionScript, you usually define an `id` property for that component in the MXML tag. For example, the following code sets the `id` property of the Button control to the String `"myButton"`:

```
<mx:Button id="myButton" label="Click Me"/>
```

This property is optional if you do not want to access the component with ActionScript.

This code causes the MXML compiler to autogenerate a public variable named `myButton` that contains a reference to that Button instance. This autogenerated variable lets you access the component instance in ActionScript. You can explicitly refer to the Button control's instance with its `id` instance reference in any ActionScript class or script block. By referring to a component's instance, you can modify its properties and call its methods.

For example, the following ActionScript block changes the value of the Button control's `label` property when the user clicks the button:

```
<?xml version="1.0"?>
<!-- usingas/ButtonExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        private function setLabel():void {
            myButton.label = "Clicked";
        }
    ]]></mx:Script>

    <mx:Button id="myButton" label="Click Me" click="setLabel();"/>

</mx:Application>
```

The IDs for all tags in an MXML component, no matter how deeply nested they are, generate public variables of the component being defined. As a result, all `id` properties must be unique within a document. This also means that if you specified an ID for a component instance, you can access that component from anywhere in the application: from functions, external class files, imported ActionScript files, or inline scripts.

You can refer to a Flex component if it does not have an `id` property by using methods of the component's container, such as the `getChildAt()` and `getChildByName()` methods.

You can refer to the current enclosing document or current object using the `this` keyword.

You can also get a reference to a component when you have a String that matches the name. To access an object on the application, you use the `this` keyword, followed by square brackets, with the String inside the square brackets. The result is a reference to the objects whose name matches the String.

The following example changes style properties on each Button control using a compound String to get a reference to the object:

```
<?xml version="1.0"?>
<!-- usingas/FlexComponents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  public function changeLabel(s:String):void {
     s = "myButton" + s;
     this[s].setStyle("fontStyle","italic");
     this[s].setStyle("fontSize","18");
  }
  ]]></mx:Script>

  <mx:Button id="myButton1" click="changeLabel('2')" label="Change Other
     Button's Styles"/>
  <mx:Button id="myButton2" click="changeLabel('1')" label="Change Other
     Button's Styles"/>
</mx:Application>
```

This technique is especially useful if you use a Repeater control or when you create objects in ActionScript and do not necessarily know the names of the objects you want to refer to prior to run time. However, when you instantiate an object in ActionScript, to add that object to the properties array, you must declare the variable as public and declare it in the class's scope, not inside a function.

The following example uses ActionScript to declare two Label controls in the application scope. During initialization, the labels are instantiated and their `text` properties are set. The example then gets a reference to the Label controls by appending the passed in variable to the String when the user clicks the Button controls.

```
<?xml version="1.0"?>
<!-- usingas/ASLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initLabels()">
  <mx:Script><![CDATA[
  import mx.controls.Label;

  public var label1:Label;
  public var label2:Label;

  // Objects must be declared in the application scope. Adds the names to
  // the application's properties array.

  public function initLabels():void {
     label1 = new Label();
     label1.text = "Change Me";

     label2 = new Label();
     label2.text = "Change Me";

     addChild(label1);
     addChild(label2);
  }

  public function changeLabel(s:String):void {
     // Create a String that matches the name of the Label control.
     s = "label" + s;
     // Get a reference to the label control using the
     // application's properties array.
     this[s].text = "Changed";
  }
  ]]></mx:Script>

  <mx:Button id="b1" click="changeLabel('2')" label="Change Other Label"/>
  <mx:Button id="b2" click="changeLabel('1')" label="Change Other Label"/>
</mx:Application>
```

Calling component methods

You can invoke the public methods of a component instance in your Flex application by using the following dot-notation syntax:

```
componentInstance.method([parameters]);
```

The following example invokes the `adjustThumb()` method when the user clicks the button, which invokes the public `setThumbValueAt()` method of the HSlider control:

```
<?xml version="1.0"?>
<!-- usingas/ComponentMethods.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        public function adjustThumb(s:HSlider):void {
            s.setThumbValueAt(0,4);
        }
    ]]></mx:Script>

    <mx:HSlider id="slider1"/>

    <mx:Button id="myButton" label="Set Thumb"
click="adjustThumb(slider1);"/>

</mx:Application>
```

To invoke a method from a child document (such as a custom MXML component), you can use the `parentApplication`, `parentDocument`, or `Application.application` properties. For more information, see Chapter 14, "Using the Application Container," on page 529.

Creating visual Flex components in ActionScript

You can use ActionScript to programmatically create visual Flex components using the `new` operator, in the same way that you create instances of any ActionScript class. The created component has default values for its properties, but it does not yet have a parent or any children (including any kind of internal DisplayObjects), and it is not yet on the display list in Flash Player, so you can't see it. After creating the component, you should use standard assignment statements to set any properties whose default values aren't appropriate.

Finally, you must add the new component to a container, by using that container's `addChild()` or `addChildAt()` method, so that it becomes part of the visual hierarchy of a Flex application. The first time that it is added to a container, a component's children are created. Children are created late in the component's life cycle so that you can set properties that can affect children as they are created.

When creating visual controls, you must import the appropriate package. In most cases, this is the mx.controls package, although you should check *Adobe Flex 2 Language Reference*.

The following example creates a Button control inside the HBox:

```
<?xml version="1.0"?>
<!-- usingas/ASVisualComponent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Button;
     public var button2:Button;

     public function createObject():void {
        button2 = new Button();
        button2.label = "Click Me";
        hb1.addChild(button2);
     }
  ]]></mx:Script>
  <mx:HBox id="hb1">
     <mx:Button label="Create Object" click="createObject()"/>
  </mx:HBox>
</mx:Application>
```

Flex creates the new child as the last child in the container. If you do not want the new child to be the last in the container, use the `addChildAt()` method to change the order. You can the `setChildIndex()` method after the call to the `addChild()` method, but this is less efficient.

You should declare an instance variable for each dynamically created component and store a reference to the newly created component in it, just as the MXML compiler does when you set an `id` property for a component instance tag. You can then access your dynamically created components in the same way as those declaratively created in MXML.

To programmatically remove a control, you can use the `removeChild()` or `removeChildAt()` methods. You can also use the `removeAllChildren()` method to remove all child controls from a container. Calling these methods does not actually delete the objects. If you do not have any other references to the child, Flash Player includes it in garbage collection at some future point. But if you stored a reference to that child on some object, the child is not removed from memory.

In some cases, you declaratively define a component with an MXML tag. You can set the `creationPolicy` property of the component's container to `none` to defer the creation of the controls inside that container. Then, to create a component that has been declared with a tag but not instantiated, you use the `createComponentFromDescriptor()` and `createComponentsFromDescriptors()` methods. These methods let you create a component programmatically rather than declaratively. For information on using the `creationPolicy` property, see Chapter 6, "Improving Startup Performance," in *Building and Deploying Flex 2 Applications*.

The only component you can pass to the `addChild()` method is a UIComponent. In other words, if you create a new object that is not a subclass of mx.core.UIComponent, you must wrap it in a UIComponent before you can attach it to a container. The following example creates a new Sprite object, which is not a subclass of UIComponent, and adds it as a child of the UIComponent before adding it to the Panel container:

```
<?xml version="1.0"?>
<!-- usingas/AddingChildrenAsUIComponents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import flash.display.Sprite;
        import mx.core.UIComponent;

        private function addChildToPanel():void {

            var circle:Sprite = new Sprite();
            circle.graphics.beginFill(0xFFCC00);
            circle.graphics.drawCircle(0, 0, 20);

            var c:UIComponent = new UIComponent();
            c.addChild(circle);
            panel1.addChild(c);

        }
    ]]></mx:Script>

    <mx:Panel id="panel1" height="100" width="100"/>

    <mx:Button id="myButton" label="Click Me" click="addChildToPanel();"/>

</mx:Application>
```

About scope

Scoping in ActionScript is largely a description of what the `this` keyword refers to at a particular point. In your application's core MXML file, you can access the Application object by using the `this` keyword. In a file defining an MXML component, `this` is a reference to the current instance of that component.

In an ActionScript class file, the `this` keyword refers to the instance of that class. In the following example, the `this` keyword refers to an instance of myClass. Because `this` is implicit, you do not have to include it, but it is shown here to illustrate its meaning.

```
class myClass {
  var _x:Number = 3;
  function get x():Number {
    return this._x;
  }
```

```
  function set x(y:Number):void {
    if (y > 0) {
      this._x = y;
    } else {
      this._x = 0;
    }
  }
}
```

However, in custom ActionScript and MXML components or external ActionScript class files, Flex executes in the context of those objects and classes, and the `this` keyword refers to the current scope and not the Application object scope.

Flex includes an `Application.application` property that you can use to access the root application. You can also use the `parentDocument` property to access the next level up in the document chain of a Flex application, or the `parentApplication` property to access the next level up in the application chain when one Application object uses a SWFLoader component to load another Application object.

If you write ActionScript in a component's event listener, the scope is not the component but rather the application. For example, the following code changes the label of the Button control to "Clicked" once the Button control is pressed:

```
<?xml version="1.0"?>
<!-- usingas/ButtonScope.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Button id="myButton" label="Click Me"
click="myButton.label='Clicked'"/>

</mx:Application>
```

Contrast the previous example with the following code:

```
<?xml version="1.0"?>
<!-- usingas/AppScope.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- The following does nothing because the app level scope does
         not have a label to set -->
    <mx:Button id="myButton" label="Click Me" click="label='Clicked'"/>

</mx:Application>
```

This code does not work because when an event listener executes, the `this` keyword does not refer to the Button instance; it is the Application or other top-level component instance. The second example attempts to set the `label` property of the Application object, not the `label` property of the Button.

Variables declared within a function are locally scoped to that function. These variables can share the same name as variables in outer scopes, and they do not affect the outer-scoped variable. If a variable is just used temporarily by a single method, make it a local variable of that method, not an instance variable. Use instance variables only for storing the state of an instance, because each instance variable will take up memory for the entire lifetime of the instance. You can refer to the outer-scoped variable with the `this.` prefix.

Comparing, including, and importing ActionScript code

To make your MXML code more readable, you can reference ActionScript files in your `<mx:Script>` tags, rather than insert large blocks of script. You can either include or import ActionScript files.

There is a distinct difference between including and importing code in ActionScript. *Including* copies lines of code from one file into another, as if they had been pasted at the position of the `include` statement. *Importing* adds a reference to a class file or package so that you can access objects and properties defined by external classes. Files that you import must be found in the source path. Files that you include must be located relative to the file using the `import` statement, or you must use an absolute path.

You use the `include` statement or the `<mx:Script source="filename">` tag to add ActionScript code to your Flex applications.

You use `import` statements in an `<mx:Script>` block to define the locations of ActionScript classes and packages that your Flex applications might use.

The following sections contain more detail on including and importing ActionScript code.

Including ActionScript files

To include ActionScript code, you reference an external ActionScript file in your `<mx:Script>` tags. At compile time, the compiler copies the entire contents of the file into your MXML application, as if you had actually typed it. As with ActionScript in an `<mx:Script>` block, ActionScript in included files can only consist of variable declarations if outside functions. Included files can also declare constants and namespaces, include other ActionScript files, import declarations, and use namespaces. You cannot define classes in included files.

Variables and functions defined in an included ActionScript file are available to any component in the MXML file. An included ActionScript file is not the same as an imported ActionScript class. Flex provides access to the included file's variables and functions, but does not add a new class, because the MXML file itself is a class.

Included ActionScript files do not need to be in the same directory as the MXML file. However, you should organize your ActionScript files in a logical directory structure.

If you are using Adobe Flex Data Services, Flex detects changes in ActionScript files using timestamps. If the file has changed since the last request, Flex regenerates the application before responding to the client. If you change the ActionScript in one of the imported ActionScript files, the next time the application is requested, the changes appear.

There are two ways to include an external ActionScript file in your Flex application:

- The `source` attribute of the `<mx:Script>` tag. This is the preferred method for including external ActionScript class files.
- The `include` statement inside `<mx:Script>` blocks.

The following sections describe these two methods of including an external ActionScript file.

Using the source attribute to include ActionScript files

You use the `source` attribute of the `<mx:Script>` tag to include external ActionScript files in your Flex applications. This provides a way to make your MXML files less cluttered and promotes code reuse across different applications.

Do not give the script file the same name as the application file. This causes a compiler error.

The following example shows the contents of the IncludedFile.as file:

```
// usingas/includes/IncludedFile.as
public function computeSum(a:Number, b:Number):Number {
  return a + b;
}
```

The following example imports the contents of the IncludedFile.as file. This file is located in the includes subdirectory.

```
<?xml version="1.0"?>
<!-- usingas/SourceInclude.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script source="includes/IncludedFile.as"/>
  <mx:TextInput id="ta1st" text="3"/>
  <mx:TextInput id="ta2nd" text="3"/>
  <mx:TextArea id="taMain"/>
  <mx:Button id="b1" label="Compute Sum"
     click="taMain.text=String(computeSum(Number(ta1st.text),
     Number(ta2nd.text)));"/>
</mx:Application>
```

The `source` attribute of the `<mx:Script>` tag supports both relative and absolute paths. For more information, see "Referring to external files that have been included" on page 71.

You cannot use the `source` attribute of an `<mx:Script>` tag and wrap ActionScript code inside that same `<mx:Script>` tag. To include a file and write ActionScript in the MXML file, use two `<mx:Script>` tags.

Using the include directive

The `include` directive is an ActionScript statement that copies the contents of the specified file into your MXML file. The `include` directive uses the following syntax:

```
include "file_name";
```

The following example includes the myfunctions.as file:

```
<?xml version="1.0"?>
<!-- usingas/AppScope.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script><![CDATA[

        include "includes/myfunctions.as";

    ]]></mx:Script>

    <mx:Button id="myButton" label="Click Me"
click="doSomething();doSomethingElse()"/>

</mx:Application>
```

You can specify only a single file for each `include` directive, but you can use any number of `include` directives. You can nest `include` directives; files with `include` directives can include files that have `include` directives.

The `include` directive supports only relative paths. For more information, see "Referring to external files that have been included" on page 71.

You can use the `include` only where multiple statements are allowed. For example, the following is not allowed:

```
if (expr)
  include "foo.as"; // First statement is guarded by IF, but rest are not.
...
```

The following is allowed:

```
if (expr) {
  include "foo.as"; // All statements inside { } are guarded by IF.
}
```

The use of curly braces ({ }) allows multiple statements because you can add multiple statements inside the braces.

Adobe recommends that you not use the `include` directive if you use a large number of included ActionScript files. You should try to break the code into separate class files where appropriate and store them in logical package structures.

Referring to external files that have been included

The `source` attribute of the `<mx:Script>` tag and the `include` directive refer to files in different ways.

The following are the valid paths to external files that are referenced in an `<mx:Script>` tag's `source` attribute:

- Flex Data Services *only:* Site-relative URLs, such as /scripts/myscript.as. A URL that begins with a slash is resolved relative to the context root of the application. The default application root is /*flex_app_root*.
- Relative URLs, such as ../myscript.as. A relative URL that does not start with a slash is resolved relative to the file that uses it. If the tag `<mx:Script source="../IncludedFile.as">` is included in "mysite/myfiles/myapp.mxml," the system searches for "mysite/IncludedFile.as".

For an ActionScript `include` directive, you can only reference relative URLs.

Flex searches the source path for imported classes and packages. Flex does not search the source path for files that are included using the `include` directive or the `source` attribute of the `<mx:Script>` tag.

Importing classes and packages

If you create many utility classes or include multiple ActionScript files to access commonly used functions, you might want to store them in a set of classes in their own package. You can import ActionScript classes and packages using the `import` statement. By doing this, you do not have to explicitly enter the fully qualified class names when accessing classes within ActionScript.

The following example imports the MyClass class in the MyPackage.Util package:

```
<?xml version="1.0"?>
<!-- usingas/AccessingPackagedClasses.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script><![CDATA[
        import MyPackage.Util.MyClass;

        private var mc:MyClass = new MyClass;

    ]]></mx:Script>

    <mx:Button id="myButton" label="Click Me"
click="myButton.label=mc.returnAString()"/>

</mx:Application>
```

In your ActionScript code, instead of referring to the class with its fully qualified package name (MyPackage.Util.MyClass), you refer to it as MyClass.

You can also use the wildcard character (*) to import the entire package. For example, the following statement imports the entire MyPackage.Util package:

```
import MyPackage.Util.*;
```

Flex searches the source path for imported files and packages, and includes only those that are used in the final SWF file.

It is not sufficient to simply specify the fully qualified class name. You should use fully qualified class names only when necessary to distinguish two classes with the same class name that reside in different packages.

If you import a class but do not use it in your application, the class is not included in the resulting SWF file's bytecode. As a result, importing an entire package with a wildcard does not create an unnecessarily large SWF file.

Techniques for separating ActionScript from MXML

This section follows a single sample application to show how it uses several different methods of separating ActionScript from the MXML. The Temperature application takes input from a single input field and uses a function to convert the input from Fahrenheit to Celsius. It then displays the resulting temperature in a Label control.

The following image shows the sample Temperature application:

In this simple application that calls a single function, there are several ways to separate MXML and ActionScript:

- "One MXML document (Event handling logic in event attribute)" on page 73
- "One MXML document (Event handling logic in <mx:Script> block)" on page 74
- "One MXML document and one ActionScript file (Event handling logic in separate script file)" on page 75

The following sections describe these methods.

One MXML document (Event handling logic in event attribute)

The following code shows the ActionScript event handling logic inside the MXML tag's `click` event:

```
<?xml version="1.0"?>
<!-- usingas/ASOneFile.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
     paddingLeft="10" paddingRight="10">
     <mx:HBox>
        <mx:Label text="Temperature in Farenheit:"/>
        <mx:TextInput id="farenheit" width="120"/>
        <mx:Button label="Convert"
           click="celsius.text=String((Number(farenheit.text)-32)/1.8);"/>
        <mx:Label text="Temperature in Celsius:"/>
     <mx:Label id="celsius" width="120" fontSize="24"/>
     </mx:HBox>
  </mx:Panel>
</mx:Application>
```

One MXML document (Event handling logic in <mx:Script> block)

In this example, the logic for the function is inside an `<mx:Script>` block in the MXML document, and is called from the MXML tag's `click` event, as the following code shows:

```
<?xml version="1.0"?>
<!-- usingas/ASScriptBlock.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     public function calculate():void {
        var n:Number = Number(farenheit.text);
        celsius.text=String((n-32)/1.8);
     }
  ]]></mx:Script>

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
     paddingLeft="10" paddingRight="10">
     <mx:HBox>
        <mx:Label text="Temperature in Farenheit:"/>
        <mx:TextInput id="farenheit" width="120"/>
        <mx:Button label="Convert" click="calculate();" />
        <mx:Label text="Temperature in Celsius:"/>
        <mx:Label id="celsius" width="120" fontSize="24"/>
     </mx:HBox>
  </mx:Panel>
</mx:Application>
```

One MXML document and one ActionScript file (Event handling logic in separate script file)

Here the function call is in an MXML event attribute, and the function is defined in a separate ActionScript file, as the following code shows:

```
<?xml version="1.0"?>
<!-- usingas/ASSourceFile.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <!-- Specify the ActionScript file that contains the function. -->
  <mx:Script source="includes/Sample3Script.as"/>

  <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
     paddingLeft="10" paddingRight="10">
     <mx:HBox>
        <mx:Label text="Temperature in Farenheit:"/>
        <mx:TextInput id="farenheit" width="120"/>
        <mx:Button label="Convert" click="calculate();"/>
        <mx:Label text="Temperature in Celsius:"/>
        <mx:Label id="celsius" width="120" fontSize="24"/>
     </mx:HBox>
  </mx:Panel>
</mx:Application>
```

The Sample3Script.as ActionScript file contains the following code:

```
// usingas/includes/Sample3Script.as
public function calculate():void {
  celsius.text=String((Number(farenheit.text)-32)/1.8);
}
```

Creating ActionScript components

You can create reusable components that use ActionScript, and reference these components in your Flex applications as MXML tags. Components created in ActionScript can contain graphical elements, define custom business logic, or extend existing Flex components. They can inherit from any components available in Flex.

Defining your own components in ActionScript has several benefits. Components let you divide your applications into individual modules that you can develop and maintain separately. By implementing commonly used logic within custom components, you can build a suite of reusable components that you can share among multiple Flex applications.

Also, you can base your custom components on the set of Flex components by extending from the Flex class hierarchy. You can create custom versions of Flex visual controls, as well as custom versions on nonvisual components, such as data validators, formatters, and effects.

For example, you can define a custom button, derived from the Button control, in the myControls package, as the following example shows:

```
package myControls {
  import mx.controls.Button;
  public class MyButton extends Button {
    public function MyButton() {
      ...
    }
    ...
  }
}
```

In this example, you write your MyButton control to the MyButton.as file, and you store the file in the myControls subdirectory of the root directory of your Flex application. The fully qualified class name of your component reflects its location. In this example, the component's fully qualified class name is myControls.MyButton.

You can reference your custom Button control from a Flex application file, such as MyApp.mxml, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cmp="myControls.*">
  <cmp:MyButton label="Jack"/>
</mx:Application>
```

In this example, you define the `cmp` namespace that defines the location of your custom component in the application's directory structure. You then reference the component as an MXML tag using the namespace prefix.

Typically, you put custom ActionScript components in directories that are in the source path. These include your application's root directory, the *flex_app_root*/WEB-INF/flex/user_classes directory (Flex Data Services only), or any directory that you specify in the `<source-path>` tag in the flex-config.xml file.

You can also create custom components using MXML. For more information, see *Creating and Extending Flex 2 Components*.

Types of custom components

You can create the following types of components in ActionScript:

User-interface components *User-interface components* contain both processing logic and visual elements. These components usually extend the Flex component hierarchy. You can extend from the UIComponent classes, or any of the Flex components, such as Button, ComboBox, or DataGrid. Your custom ActionScript component inherits all of the public methods and public and protected properties of its base class.

Nonvisual components *Nonvisual components* define no visual elements. A nonvisual component is an ActionScript class that does not extend the UIComponent class. They can provide greater efficiency at run time.

Performing object introspection

Object introspection is a technique for determining the elements of a class at run time, such as its properties and methods. There are two ways to do introspection in ActionScript:

- Using `for..in` loops
- Using the introspection API

This section describes how to use both these methods.

You might find object introspection a useful technique when debugging your application. For example, you might write a method that takes a generic object of type Object as an argument. You can use introspection to output all of the properties and methods of the Object to determine exactly what your application passed to it.

Using for..in loops

A `for..in` loop enumerates only dynamically added properties. Declared variables and methods of classes are not enumerated in `for..in` loops. This means that most classes in the ActionScript API will not display any properties in a `for..in` loop. The generic type Object is still a dynamic object and will display properties in a `for..in` loop.

The following example creates a generic Object, adds properties to that object, and then iterates over that object when you click the button to inspect its properties:

```
<?xml version="1.0"?>
<!-- usingas/IntrospectionForIn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     private var obj:Object = new Object();

     private function initApp():void {
        // Create the object.
        obj.a = "Schotten Totten";
        obj.b = "Taj Majal";
        obj.c = "Durche die Wuste";
     }

     public function dumpObj():void {
        for (var p:String in obj) {
           ta1.text += p + ":" + obj[p] + "\n";
        }
     }
  ]]></mx:Script>
  <mx:TextArea id="ta1" width="400" height="500"/>
  <mx:Button label="Dump Object" click="dumpObj()"/>
</mx:Application>
```

You can also use the `mx.utils.ObjectUtil.toString()` method to print all the dynamically added properties of an object; for example:

```
<?xml version="1.0"?>
<!-- usingas/IntrospectionForIn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.utils.ObjectUtil;

     private var obj:Object = new Object();

     private function initApp():void {
        // Create the object.
        obj.a = "Schotten Totten";
        obj.b = "Taj Majal";
        obj.c = "Durche die Wuste";
     }

     public function dumpObj():void {
        ta1.text = ObjectUtil.toString(obj);
     }
  ]]></mx:Script>
  <mx:TextArea id="ta1" width="400" height="500"/>
  <mx:Button label="Dump Object" click="dumpObj()"/>
</mx:Application>
```

The mx.utils.ObjectUtil class has other useful methods such as `compare()`, `copy()`, and `isSimple()`. For more information, see *Adobe Flex 2 Language Reference*.

Using the introspection API

If you want to list all the public properties and methods of a nondynamic (or sealed) class or class instance, use the `describeType()` method and parse the results using the E4X API. The `describeType()` method is in the flash.utils package. The method's only parameter is the target object that you want to introspect. You can pass it any ActionScript value, including all available ActionScript types such as object instances, primitive types such as `uint`, and class objects. The return value of the `describeType()` method is an E4X XML object that contains an XML description of the object's type.

The `describeType()` method returns only public members. The method does not return private members of the caller's superclass or any other class where the caller is not an instance. If you call `describeType(this)`, the method returns information only about nonstatic members of the class. If you call `describeType(getDefinitionByName("MyClass"))`, the method returns information only about the target's static members.

The following example introspects the Button control and prints the details to TextArea controls:

```
<?xml version="1.0"?>
<!-- usingas/IntrospectionAPI.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="getDetails()">
    <mx:Script><![CDATA[
        import flash.utils.*;

        public function getDetails():void {
            // Get the Button control's E4X XML object description.
            var classInfo:XML = describeType(button1);

            // Dump the entire E4X XML object into ta2.
            ta2.text = classInfo.toString();

            // List the class name.
            ta1.text = "Class " + classInfo.@name.toString() + "\n";

            // List the object's variables, their values, and their types.
            for each (var v:XML in classInfo..variable) {
               ta1.text += "Variable " + v.@name + "=" + button1[v.@name] +
                    " (" + v.@type + ")\n";
            }

            // List accessors as properties.
            for each (var a:XML in classInfo..accessor) {
                // Do not get the property value if it is write only.
                if (a.@access == 'writeonly') {
                  ta1.text += "Property " + a.@name + " (" + a.@type +")\n";
                }
                else {
                    ta1.text += "Property " + a.@name + "=" +
                        button1[a.@name] +  " (" + a.@type +")\n";
                }
            }

            // List the object's methods.
            for each (var m:XML in classInfo..method) {
                ta1.text += "Method " + m.@name + "():" + m.@returnType +
"\n";
            }
        }
    ]]></mx:Script>

    <mx:Button label="Submit" id="button1"/>
    <mx:TextArea id="ta1" width="400" height="200"/>
    <mx:TextArea id="ta2" width="400" height="200"/>
</mx:Application>
```

The output displays accessors, variables, and methods of the Button control, and appears similar to the following:

```
Class mx.controls::Button
...
Variable id=button1 (String)
Variable __width=66 (Number)
Variable layoutWidth=66 (Number)
Variable __height=22 (Number)
Variable layoutHeight=22 (Number)
...
Property label=Submit (String)
Property enabled=true (Boolean)
Property numChildren=2 (uint)
Property enabled=true (Boolean)
Property visible=true (Boolean)
Property toolTip=null (String)
...
Method dispatchEvent():Boolean
Method hasEventListener():Boolean
Method layoutContents():void
Method getInheritingStyle():Object
Method getNonInheritingStyle():Object
```

Another useful method is the ObjectUtil's `getClassInfo()` method. This method returns an Object with the name and properties of the target object. The following example uses the `getClassInfo()` and `toString()` methods to show the properties of the Button control:

```
<?xml version="1.0"?>
<!-- usingas/IntrospectionObjectUtil.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     import mx.utils.ObjectUtil;
     private function showProps(b:Button):void {
        var o:Object = ObjectUtil.getClassInfo(b);
        ta1.text = ObjectUtil.toString(o);
     }
  ]]></mx:Script>
  <mx:Button id="b1" label="Show Properties" click="showProps(b1)"/>
  <mx:TextArea id="ta1" width="300" height="500"/>
</mx:Application>
```

For more information about using E4X, see *Programming ActionScript 3.0*.

CHAPTER 5

Using Events

5

One of the most important parts of your Adobe Flex application is handling events. This topic describes the event flow and how to handle events by using controls and ActionScript in your Flex applications.

Contents

About events

This section introduces you to the event model in Flex 2. In addition, this section describes the Event object and its subclasses, and describes the event dispatching model. For a quick start in using events in Flex, you can skip this section and see sample code in "Using events" on page 87.

Events let a developer know when something happens within a Flex application. They can be generated by user devices, such as the mouse and keyboard, or other external input, such as the return of a web service call. Events are also triggered when changes happen in the appearance or life cycle of a component, such as the creation or destruction of a component or when the component is resized.

Any user interaction with your application can generate events. Events can also occur without any direct user interaction, such as when data finishes loading from a server or when an attached camera becomes active. You can "handle" these events in your code by adding an event handler. *Event handlers* are the functions or methods that you write to respond to specific events. They are also sometimes referred to as *event listeners*.

The Flex event model is based on the Document Object Model (DOM) Level 3 Events Model. Although Flex does not adhere specifically to the Document Object Model standard, the implementations are very similar.

Components generate and dispatch events and *consume* (listen to) other events. An object that requires information about another object's events registers a listener with that object. When an event occurs, the object dispatches the event to all registered listeners by calling a function that was requested during registration. To receive multiple events from the same object, you must register your listener for each event.

Components have built-in events that you can handle in ActionScript blocks in your MXML applications. You can also take advantage of the Flex event system's dispatcher-listener model to define your own event listeners outside of your applications, and define which methods of your custom listeners will listen to certain events. You can register listeners with the target object so that when the target object dispatches an event, the listeners get called.

All visual objects, including Flex controls and containers, are subclasses of the DisplayObject class. They are in a tree of visible objects that make up your application. The root of the tree is the Stage. Below that is the SystemManager object, and then the Application object. Child containers and components are leaf nodes of the tree. That tree is known as the *display list*. An object on the display list is analogous to a node in the DOM hierarchical structure. The terms *display list object* and *node* are used interchangeably in this topic.

For information about each component's events, see the component's description in Chapter 9, "Using Controls," on page 259 or the control's entry in *Adobe Flex 2 Language Reference*.

For a detailed description of a component's startup life cycle, including major events in that life cycle, see Chapter 10, "Creating Advanced Visual Components in ActionScript," in *Creating and Extending Flex 2 Components*.

About the Event flow

You can instruct any container or control to listen for events dispatched by another container or control. When Adobe Flash Player dispatches an Event object, that Event object makes a round-trip journey from the root of the display list to the target node, checking each node for registered listeners. The *target node* is the node in the display list where the event occurred. For example, if a user clicks a Button control named Child1, Flash Player dispatches an Event object with Child1 defined as the target node.

The event flow is conceptually divided into three parts. The following sections introduce you to these parts. For more information about the event flow, see "Event propagation" on page 112.

About the capturing phase

The first part of the event flow is called the *capturing phase*, which comprises all of the nodes from the root node to the parent of the target node. During this phase, Flash Player examines each node, starting with the root, to see if it has a listener registered to handle the event. If it does, Flash Player sets the appropriate values of the Event object and then calls that listener. Flash Player stops after it reaches the target node's parent and calls any listeners registered on the parent. For more information about the capturing phase, see "Capturing phase" on page 114.

About the targeting phase

The second part of the event flow is called the *targeting phase*, which consists solely of the target node. Flash Player sets the appropriate values on the Event object, checks the target node for registered event listeners, and then calls those listeners. For more information about the targeting phase, see "Targeting phase" on page 115.

About the bubbling phase

The third part of the event flow is called the *bubbling phase*, which comprises all of the nodes from the target node's parent to the root node. Starting with the target node's parent, Flash Player sets the appropriate values on the Event object and then calls event listeners on each of these nodes. Flash Player stops after calling any listeners on the root node. For more information about the bubbling phase, see "Bubbling phase" on page 115.

About the Event class

The flash.events.Event class is an ActionScript class with properties that contain information about the event that occurred. An Event object is an implicitly created object, similar to the way the request and response objects in a JavaServer Page (JSP) are implicitly created by the application server.

Flex creates an Event object each time an event is dispatched. You can use the Event object inside an event listener to access details about the event that was dispatched, or about the component that dispatched the event. Passing an Event object to, and using it in, an event listener is optional. However, if you want to access the Event object's properties inside your event listeners, you must pass the Event object to the listener.

Flex creates only one Event object when an event is dispatched. During the bubbling and capturing phases, Flex changes the values on the Event object as it moves up or down the display list, rather than creating a new Event object for each node.

About event subclasses

There are many classes that extend the flash.events.Event class. These classes are defined mostly in the following two packages:

- mx.events.*
- flash.events.*

The mx.events package defines event classes that are specific to Flex controls, including the DataGridEvent, DragEvent, and ColorPickerEvent. The flash.events package describes events that are not unique to Flex but are instead defined by Flash Player. These event classes include MouseEvent, DataEvent, and TextEvent. All of these events are commonly used in Flex applications.

In addition to these packages, some packages also define their own event objects; for example, mx.messaging.events.ChannelEvent and mx.logging.LogEvent.

Child classes of the Event class have additional properties and methods that may be unique to them. In some cases you will want to use a more specific event type rather than the generic Event object so that you can access these unique properties or methods. For example, the LogEvent class has a `getLevelString()` method that the Event class does not.

For information on using Event subclasses, see "Using event subclasses" on page 121.

About the EventDispatcher class

Every object in the display list can trace its class inheritance back to the DisplayObject class. The DisplayObject class, in turn, inherits from the EventDispatcher class. The EventDispatcher class is a base class that provides important event model functionality for every object on the display list. Because the DisplayObject class inherits from the EventDispatcher class, any object on the display list has access to the methods of the EventDispatcher class.

This is significant because every item on the display list can participate fully in the event model. Every object on the display list can use its `addEventListener()` method—inherited from the EventDispatcher class—to listen for a particular event, but only if the listening object is part of the event flow for that event.

Although the name EventDispatcher seems to imply that this class's main purpose is to send (or dispatch) Event objects, the methods of this class are used much more frequently to register event listeners, check for event listeners, and remove event listeners.

The EventDispatcher class implements the IEventDispatcher interface. This allows developers who create custom classes that cannot inherit from EventDispatcher or one of its subclasses to implement the IEventDispatcher interface to gain access to its methods.

The `addEventListener()` method is the most commonly used method of this class. You use it to register your event listeners. For information on using the `addEventListener()` method, see "Using the addEventListener() method" on page 95.

Advanced programmers use the `dispatchEvent()` method to manually dispatch an event or to send a custom Event object into the event flow. For more information, see "Manually dispatching events" on page 109.

Several other methods of the EventDispatcher class provide useful information about the existence of event listeners. The `hasEventListener()` method returns `true` if an event listener is found for that specific event type on a particular display list object. The `willTrigger()` method checks for event listeners on a particular display list object, but it also checks for listeners on all of that display list object's ancestors for all phases of the event flow. The method returns `true` if it finds one.

Using events

Using events in Flex is a two-step process. First, you write a function or class method, known as an *event listener* or *event handler,* that responds to events. The function often accesses the properties of the Event object or some other settings of the application state. The signature of this function usually includes an argument that specifies the event type being passed in.

The following example shows a simple event listener function that reports when a control triggers the event that it is listening for:

```
<?xml version="1.0"?>
<!-- events/SimpleEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.controls.Alert;

     private function initApp():void {
        b1.addEventListener(MouseEvent.CLICK, myEventHandler);
     }

     private function myEventHandler(event:Event):void {
        Alert.show("An event occurred");
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>

</mx:Application>
```

As you can see in this example, you also register that function or class method with a display list object by using the `addEventListener()` method.

Most Flex controls simplify listener registration by letting you specify the listener inside the MXML tag. For example, instead of using the `addEventListener()` method to specify a listener function for the Button control's `click` event, you specify it in the `click` attribute of the `<mx:Button>` tag:

```
<?xml version="1.0"?>
<!-- events/SimplerEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Alert;

     private function myEventHandler(event:Event):void {
        Alert.show("An event occurred");
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>

</mx:Application>
```

This is equivalent to the `addEventListener()` method in the previous code example. However, it is best practice to use the `addEventListener()` method. This method gives you greater control over the event by letting you configure the priority and capturing settings, and use event constants. In addition, if you use `addEventListener()` to add an event handler, you can use `removeEventListener()` to remove the handler when you no longer need it. If you add an event handler inline, you cannot call `removeEventListener()` on that handler.

Each time a control generates an event, Flex creates an Event object that contains information about that event, including the type of event and a reference to the dispatching control. To use the Event object, you specify it as a parameter in the event handler function, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/EventTypeHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Alert;

     private function myEventHandler(e:Event):void {
        Alert.show("An event of type " + e.type + " occurred.");
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>

</mx:Application>
```

If you want to access the Event object in an event handler that was triggered by an inline event, you must add the `event` keyword inside the MXML tag so that Flex explicitly passes it to the handler, as in the following:

```
<mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
```

You are not required to use the Event object in a handler function. The following example creates two event handler functions and registers them with the events of a ComboBox control. The first event handler, `openEvt()`, takes no arguments. The second event handler, `changeEvt()`, takes the Event object as an argument and uses this object to access the `value` and `selectedIndex` of the ComboBox control that triggered the event.

```
<?xml version="1.0"?>
<!-- events/MultipleEventHandlers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     private function openEvt():void {
        forChange.text="";
     }
     private function changeEvt(e:Event):void {
        forChange.text=e.currentTarget.value + " " +
           e.currentTarget.selectedIndex;
     }
  ]]></mx:Script>
  <mx:ComboBox open="openEvt()" change="changeEvt(event)">
     <mx:dataProvider>
        <mx:Array>
           <mx:String>AK</mx:String>
           <mx:String>AL</mx:String>
           <mx:String>AR</mx:String>
        </mx:Array>
     </mx:dataProvider>
  </mx:ComboBox>
  <mx:TextArea id="forChange" width="150"/>
</mx:Application>
```

This example shows accessing the `target` property of the Event object. For more information, see "Accessing the target property" on page 91.

Specifying the Event object

You specify the object in a listener function's signature as type Event, as the following example shows:

```
function myEventListener(e:Event):void { ... }
```

However, if you want to access properties that are specific to the type of event that was dispatched, you must instead specify a more specific event type, such as ToolTipEvent or KeyboardEvent, as the following example shows:

```
import mx.events.ToolTip
function myEventListener(e:ToolTipEvent):void { ... }
```

In some cases, you must import the event's class in your ActionScript block.

Most objects have specific events that are associated with them, and most of them can dispatch more than one type of event.

If you declare an event of type Event, you can cast it to a more specific type to access its event-specific properties. For more information, see "Using event subclasses" on page 121.

Accessing the target property

Event objects include a reference to the instance of the dispatching component (or *target*). So, you can access all the properties and methods of that instance in an event listener. The following example accesses the `id` of the button control that triggered the event:

```
<?xml version="1.0"?>
<!-- events/AccessingCurrentTarget.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     private function myEventHandler(e:Event):void {
        Alert.show("The button " + e.currentTarget.id + " was clicked");
     }
]]></mx:Script>
  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</mx:Application>
```

Calling methods and accessing properties on the current target can be confusing. The type of the `currentTarget` property is DisplayObject. Because ActionScript is strongly typed, you can call `event.currentTarget.`*`methodName`*`()` only if *`methodName`* is a method defined on DisplayObject. The same applies for properties. You can access `event.currentTarget.`*`property`* only if the *`property`* is defined on DisplayObject. If you try to call another method on the `currentTarget` (for example, the `setStyle()` method), Flex returns an error. The `setStyle()` method is defined on UIComponent, a subclass of DisplayObject. Therefore, you must cast `currentTarget` to UIComponent before calling the `setStyle()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/InvokingOnCurrentTarget.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.core.UIComponent;

     private function myEventHandler(e:Event):void {
        UIComponent(e.currentTarget).setStyle("color", "red");
     }
  ]]></mx:Script>
  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>
</mx:Application>
```

You can also access methods and properties of the `target`, which contains a reference to the current node in the display list. For more information, see "About the target and currentTarget properties" on page 113.

Registering event handlers

There are several strategies that you can employ when you register event handlers with your Flex controls:

- Define an event handler inline. This binds a call to the handler function to the control that triggers the event.

```
<?xml version="1.0"?>
<!-- events/SimplerEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Alert;

     private function myEventHandler(event:Event):void {
        Alert.show("An event occurred");
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>

</mx:Application>
```

  In this example, whenever the user clicks the Button control, Flex calls the `myClickHandler()` function.

  For more information on defining event handlers inline, see "Defining event listeners inline" on page 93.

- Use the `addEventListener()` method, as follows:

```
<?xml version="1.0"?>
<!-- events/SimpleEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.controls.Alert;

     private function initApp():void {
        b1.addEventListener(MouseEvent.CLICK, myEventHandler);
     }

     private function myEventHandler(event:Event):void {
        Alert.show("An event occurred");
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>

</mx:Application>
```

  As with the previous example, whenever the user clicks the Button control, Flex calls the `myClickHandler()` handler function. But registering your event handlers using this method provides more flexibility. You can register multiple components with this event handler, add multiple handlers to a single component, or remove the handler. For more information, see "Using the addEventListener() method" on page 95.

- Create an event handler class and register components to use the class for event handling. This approach to event handling promotes code reuse and lets you centralize event handling outside your MXML files. For more information on creating custom event handler classes, see "Creating event handler classes" on page 99.

The following sections describe these methods of handling events.

Defining event listeners inline

The simplest method of defining event handlers in Flex applications is to point to a handler function in the component's MXML tag. To do this, you add any of the component's events as a tag attribute followed by an ActionScript statement or function call.

You add an event handler inline using the following syntax:

`<mx:tag_name event_name="handler_function"/>`

For example, to listen for a Button control's `click` event, you add a statement in the `<mx:Button>` tag's `click` attribute. If you add a function, you define that function in an ActionScript block. The following example defines the `submitForm()` function as the handler for the Button control's `click` event:

```
<mx:Script><![CDATA[
  function submitForm():void {
    // Do something.
  }
]]></mx:Script>
<mx:Button label="Submit" click="submitForm();" />
```

Event handlers can include any valid ActionScript code, including code that calls global functions or sets a component property to the return value. The following example calls the `trace()` global function:

```
<mx:Button label="Get Ver" click="trace('The button was clicked');"/>
```

There is one special parameter that you can pass in an inline event handler definition: the `event` parameter. If you add the `event` keyword as a parameter, Flex passes the Event object. Inside the handler function, you can then access all the properties of the Event object.

The following example passes the Event object to the `submitForm()` handler function and specifies it as type MouseEvent:

```
<?xml version="1.0"?>
<!-- events/MouseEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     private function myEventHandler(event:MouseEvent):void {
        // Do something with the MouseEvent object;
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me" click="myEventHandler(event)"/>

</mx:Application>
```

It is best practice to include the `event` keyword when you define all inline event listeners and to specify the most stringent Event object type in the resulting listener function (for example, specify MouseEvent instead of Event).

You can use the Event object to access a reference to the target object (the object that dispatched the event), the type of event (for example, `click`), or other relevant properties, such as the row number and value in a list-based control. You can also use the Event object to access methods and properties of the target component, or the component that dispatched the event.

Although you will most often pass the entire Event object to an event listener, you can just pass individual properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/PropertyHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     private function myEventHandler(s:String):void {
        trace(s);
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me"
click="myEventHandler(event.currentTarget.id)"/>

</mx:Application>
```

Registering an event listener inline provides less flexibility than using the `addEventListener()` method to register event listeners. The drawbacks are that you cannot set the `useCapture` or `priority` properties on the Event object and that you cannot remove the listener once you add it.

Using the addEventListener() method

The `addEventListener()` method lets you register event listener functions with the specified control or object. The following example adds the `myClickListener()` function to the b1 instance of a Button control. When the user clicks b1, Flex calls the `myClickListener()` method:

```
b1.addEventListener(MouseEvent.CLICK, myClickListener);
```

The `addEventListener()` method has the following signature:

```
componentInstance.addEventListener(event_type:String,
  event_listener:Function, use_capture:Boolean, priority:int,
  weakRef:Boolean)
```

The *`event_type`* argument is the kind of event that this component dispatches. This can be either the event type String (for example, `click` or `mouseOut`), or the event type static constant (such as `MouseEvent.CLICK` or `MouseEvent.MOUSE_OUT`). This argument is required.

The constants provide an easy way to refer to specific event types. You should use these constants instead of the strings that they represent. If you misspell a constant name in your code, the compiler catches the mistake. If you instead use strings and make a typographical error, it can be harder to debug and could lead to unexpected behavior.

You should use the constants wherever possible. For example, when you are testing to see whether an Event object is of a certain type, use the following code:

```
if (myEventObject.type == MouseEvent.CLICK) {/* your code here */}
```

rather than:

```
if (myEventObject.type == "click") {/* your code here */}
```

The *event_listener* argument is the function that handles the event. This argument is required.

The *use_capture* parameter of the `addEventListener()` method lets you control the phase in the event flow in which your listener will be active. It sets the value of the `useCapture` property of the Event object. If `useCapture` is set to `true`, your listener is active during the capturing phase of the event flow. If `useCapture` is set to `false`, your listener is active during the targeting and bubbling phases of the event flow but not during the capturing phase. The default value is determined by the type of event, but is `false` in most cases.

To listen for an event during all phases of the event flow, you must call `addEventListener()` twice, once with the *use_capture* parameter set to `true`, and again with *use_capture* set to `false`. This argument is optional. For more information, see "Capturing phase" on page 114.

The *priority* parameter sets the priority for that event listener. The higher the number, the sooner that event handler executes relative to other event listeners for the same event. Event listeners with the same priority are executed in the order that they were added. This parameter sets the `priority` property of the Event object. The default value is 0, but you can set it to negative or positive integer values. If several event listeners were added without priorities, the earlier a listener is added, the sooner it is executed. For more information on setting priorities, see "Event priorities" on page 119.

The *weakRef* parameter provides you with some control over memory resources for listeners. A strong reference (when `weakRef` is `false`) prevents the listener from being garbage collected. A weak reference (when `weakRef` is `true`) does not. The default value is `false`.

When you add a listener function and that function is invoked, Flex implicitly creates an Event object for you and passes it to the listener function. You must declare the Event object in the signature of your listener function.

If you add an event listener by using the `addEventListener()` method, you are required to declare an event object as a parameter of the *listener_function*, as the following example shows:

```
b1.addEventListener(MouseEvent.CLICK, performAction);
```

In the listener function, you declare the Event object as a parameter, as follows:

```
public function performAction(e:MouseEvent):void {
    ...
}
```

The following example defines a new handler function `myClickListener()`. It then registers the `click` event of the Button control with that handler. When the user clicks the button, Flex calls the `myClickHandler()` function.

```
<?xml version="1.0"?>
<!-- events/AddEventListenerExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="createListener()">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     private function createListener():void {
        b1.addEventListener(MouseEvent.CLICK, myClickHandler, false, 0);
     }
     private function myClickHandler(e:MouseEvent):void {
        Alert.show("The button was clicked");
     }
  ]]></mx:Script>
  <mx:Button label="Click Me" id="b1"/>
</mx:Application>
```

Using addEventListener() inside an MXML tag

You can add event listeners with the `addEventListener()` method inline with the component definition. The following Button control definition adds the call to the `addEventListener()` method inline with the Button control's `initialize` property:

```
<?xml version="1.0"?>
<!-- events/CallingAddEventListenerInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.controls.Alert;

  private function myClickHandler(event:Event):void {
     Alert.show("This is a log message");
  }
  ]]></mx:Script>

  <mx:Button id='b1'
     label="Click Me"
     initialize='b1.addEventListener(MouseEvent.CLICK, myClickHandler,
false, 1);'
  />

</mx:Application>
```

This is the equivalent of defining the event handler inline. However, defining a handler by using the `addEventListener()` method rather than setting `click="handler_function"` lets you set the value of the `useCapture` and `priority` properties of the Event object. Furthermore, you cannot remove a handler added inline, but when you use the `addEventListener()` method to add a handler, you can call the `removeEventListener()` method to remove that handler.

Removing event handlers

It is a good idea to remove any handlers that will no longer be used. This removes references to objects so that they can be cleared from memory. You can use the `removeEventListener()` method to remove an event handler that you no longer need. All components that can call `addEventListener()` can also call the `removeEventListener()` method. The syntax for the `removeEventListener()` method is as follows:

```
componentInstance.removeEventListener(event_type:String,
  listener_function:Function, use_capture:Boolean)
```

For example, consider the following code:

```
myButton.removeEventListener(MouseEvent.CLICK, myClickHandler);
```

The *`event_type`* and *`listener_function`* parameters are required. These are the same as the required parameters for the `addEventListener()` method.

The *`use_capture`* parameter is also identical to the parameter used in the `addEventListener()` method. Recall that you can listen for events during all event phases by calling `addEventListener()` twice; once with *`use_capture`* set to `true`, and again with it set to `false`. To remove both event listeners, you must call `removeEventListener()` twice; once with *`use_capture`* set to `true`, and again with it set to `false`.

You can only remove event listeners that you added with the `addEventListener()` method in an ActionScript block. You cannot remove an event listener that was defined in the MXML tag, even if it was registered using a call to the `addEventListener()` method that was made inside a tag attribute.

The following sample application shows what type of handler can be removed and what type cannot:

```
<?xml version="1.0"?>
<!-- events/RemoveEventListenerExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="createHandler(event)">
  <mx:Script><![CDATA[
  import mx.controls.Alert;
     private function createHandler(e:Event):void {
        b1.addEventListener(MouseEvent.CLICK, myClickHandler);
     }
     private function removeMyHandlers(e:Event):void {
        // Remove listener for b1's click event.
        b1.removeEventListener(MouseEvent.CLICK,myClickHandler);

        // Does NOT remove the listener for b2's click event.
        b2.removeEventListener(MouseEvent.CLICK,myClickHandler);
     }
     private function myClickHandler(e:Event):void {
        Alert.show("This is a log message");
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me"/>
  <mx:Button label="Click Me Too" id="b2" click="myClickHandler(event)"/>
  <mx:Button label="Axe Listeners" id="b3" click="removeMyHandlers(event)"/
>
</mx:Application>
```

Creating event handler classes

You can create an external class file and use the methods of this class as event handlers. Objects themselves cannot be event handlers, but methods of an object can be. By defining one class that handles all your event handlers, you can use the same event handling logic across applications, which can make your MXML applications more readable and maintainable.

To create a class that handles events, you usually import the flash.events.Event class. You also usually write an empty constructor. The following ActionScript class file calls the `trace()` function whenever it handles an event with the `handleAllEvents()` method:

```
// events/MyEventHandler.as

package { // Empty package

  import flash.events.Event;

  public class MyEventHandler {
     public function MyEventHandler() {
        // Empty constructor
     }

     public function handleAllEvents(event:Event):void {
        trace("some event happened");
     }
  }
}
```

In your MXML file, you declare a new instance of MyEventHandler and use the `addEventListener()` method to register its `handleAllEvents()` method as a handler to the Button control's `click` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/CustomHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="createHandler()">
  <mx:Script><![CDATA[
     private var myListener:MyEventHandler = new MyEventHandler();

     private function createHandler():void {
        b1.addEventListener(MouseEvent.CLICK, myListener.handleAllEvents);
  }
  ]]></mx:Script>

  <mx:Button label="Submit" id="b1"/>

</mx:Application>
```

The best approach is to define the event handler's method as static. By doing this, you are no longer required to instantiate the class inside your MXML application. The following `createHandler()` function registers the `handleAllEvents()` method as an event handler without instantiating the MyStaticEventHandler class:

```
<?xml version="1.0"?>
<!-- events/CustomHandlerStatic.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="createHandler()">
  <mx:Script><![CDATA[
     private function createHandler():void {
        b1.addEventListener(MouseEvent.CLICK,
MyStaticEventHandler.handleAllEvents);
  }
  ]]></mx:Script>

  <mx:Button label="Submit" id="b1"/>

</mx:Application>
```

In the class file, you just add the `static` keyword to the method signature:

```
// events/MyStaticEventHandler.as

package { // Empty package

  import flash.events.Event;

  public class MyStaticEventHandler {
     public function MyStaticEventHandler() {
        // Empty constructor
     }

     public static function handleAllEvents(event:Event):void {
        trace("some event happened");
     }
  }
}
```

Store your event listener class in a directory in your source path. You can also store your ActionScript class in the same directory as your MXML file, although Adobe does not recommend this.

Defining multiple listeners for a single event

You can define multiple event handler functions for a single event in two ways. When defining events inside MXML tags, you separate each new handler function with a semicolon. The following example adds the `submitForm()` and `debugMessage()` functions as handlers of the `click` event:

```
<?xml version="1.0"?>
<!-- events/MultipleEventHandlersInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     private function submitForm(e:Event):void {
        // Handle event here.
     }
     private function debugMessage(e:Event):void {
        // Handle event here.
     }
  ]]></mx:Script>

  <mx:Button id="b1"
     label="Do Both Actions"
     click='submitForm(event); debugMessage(event);'
  />

</mx:Application>
```

For events added with the `addEventListener()` method, you can add any number of handlers with additional calls to the `addEventListener()` method. Each call adds a handler function that you want to register to the specified object. The following example registers the `submitForm()` and `debugMessage()` handler functions with b1's `click` event:

```
<?xml version="1.0"?>
<!-- events/MultipleEventHandlersAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="createHandlers(event)">
  <mx:Script><![CDATA[

     public function createHandlers(e:Event):void {
        b1.addEventListener(MouseEvent.CLICK, submitForm);
        b1.addEventListener(MouseEvent.CLICK, debugMessage);
     }

     private function submitForm(e:Event):void {
        // Handle event here.
     }
     private function debugMessage(e:Event):void {
        // Handle event here.
     }

  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me"/>

</mx:Application>
```

You can mix the methods of adding event handlers to any component; alternatively, you can add handlers inline and with the `addEventListener()` method. The following example adds a `click` event handler inline for the Button control which calls the `performAction()` method. It then conditionally adds a second `click` handler to call the `logAction()` method, depending on the state of the CheckBox control.

```
<?xml version="1.0"?>
<!-- events/ConditionalHandlers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initApp(event)">
  <mx:Script><![CDATA[
     import mx.controls.Alert;

     private function initApp(e:Event):void {
        cb1.addEventListener(MouseEvent.CLICK, handleCheckBoxChange);
        b1.addEventListener(MouseEvent.CLICK, logAction);
     }

     private function handleCheckBoxChange(e:Event):void {
        if (cb1.selected) {
           b1.addEventListener(MouseEvent.CLICK, logAction);
           ta1.text += "added log listener" + "\n";
        } else {
           b1.removeEventListener(MouseEvent.CLICK, logAction);
           ta1.text += "removed log listener" + "\n";
        }
     }

     private function performAction(e:Event):void {
        Alert.show("You performed the action");
     }

     private function logAction(e:Event):void {
        ta1.text += "Action performed: " + e.type + "\n";
     }
  ]]></mx:Script>

  <mx:Button label="Perform Action" id="b1" click="performAction(event)"/>
  <mx:CheckBox id="cb1" label="Log?" selected="true"/>
  <mx:TextArea id="ta1" height="200" width="300"/>

</mx:Application>
```

You can set the order in which event listeners are called by using the `priority` parameter of the `addEventListener()` method. You cannot set a priority for a listener function if you added the event listener using MXML inline. For more information on setting priorities, see "Event priorities" on page 119.

Registering a single listener with multiple components

You can register the same listener function with any number of events of the same component, or events of different components. The following example registers a single listener function, `submitForm()`, with two different buttons:

```
<?xml version="1.0"?>
<!-- events/OneHandlerTwoComponentsInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[

     private function submitForm(e:Event):void {
        // Handle event here.
        trace(e.currentTarget.id);
     }

  ]]></mx:Script>

  <mx:Button id="b1"
     label="Click Me"
     click="submitForm(event)"
  />

  <mx:Button id="b2"
     label="Click Me"
     click="submitForm(event)"
  />

</mx:Application>
```

When you use the `addEventListener()` method to register a single listener to handle the events of multiple components, you must use a separate call to the `addEventListener()` method for each instance, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/OneHandlerTwoComponentsAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="createHandlers(event)">
  <mx:Script><![CDATA[

     public function createHandlers(e:Event):void {
        b1.addEventListener(MouseEvent.CLICK, submitForm);
        b2.addEventListener(MouseEvent.CLICK, submitForm);
     }

     private function submitForm(e:Event):void {
        // Handle event here.
        trace(e.currentTarget.id);
     }

  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me"/>

  <mx:Button id="b2" label="Click Me Too"/>

</mx:Application>
```

When doing this, you should add logic to the event listener that processes the type of event. The event target (or object that dispatched the event) is added to the Event object for you. No matter what triggered the event, you can conditionalize the event processing based on the `target` or `type` properties of the Event object. Flex adds these two properties to all Event objects.

The following example registers a single listener function (`myEventHandler()`) to the `click` event of a Button control and the `click` event of a CheckBox control. To detect what type of object called the event listener, the listener checks the `className` property of the target in the Event object in a `case` statement:

```
<?xml version="1.0"?>
<!-- events/ConditionalTargetHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     public function initApp():void {
        button1.addEventListener(MouseEvent.CLICK, myEventHandler);
        cb1.addEventListener(MouseEvent.MOUSE_DOWN, myEventHandler);
     }

     public function myEventHandler(event:Event):void {
        switch (event.currentTarget.className) {
           case "Button":
            // Process Button click.
            break;
           case "CheckBox":
            // Process CheckBox click.
            break;
        }
     }
  ]]></mx:Script>

  <mx:Button label="Submit" id="button1"/>
  <mx:CheckBox label="All Words" id="cb1"/>
  <mx:TextArea id="ta1" text="Please enter a search term" width="200"/>

</mx:Application>
```

Passing additional parameters to listener functions

You can pass additional parameters to listener functions depending on how you add the listeners. If you add a listener with the `addEventListener()` method, you cannot pass any additional parameters to the listener function, and that listener function can only declare a single argument, the Event object (or one of its subclasses).

For example, the following code throws an error because the `clickListener()` method expects two arguments:

```
<mx:Script>
  public function addListeners():void {
    b1.addEventListener(MouseEvent.CLICK,clickListener);
  }
  public function clickListener(e:MouseEvent, a:String):void { ... }
</mx:Script>
<mx:Button id="b1"/>
```

Because the second parameter of `addEventListener()` is a function, you cannot specify parameters of that function in the `addEventListener()` call. So, to pass additional parameters to the listener function, you must define them in the listener function and then call the final method with those parameters. If you define an event listener inline (inside the MXML tag), you can add any number of parameters as long as the listener function's signature agrees with that number of parameters. The following example passes a string and the Event object to the `runMove()` method:

```
<?xml version="1.0"?>
<!-- events/MultipleHandlerParametersInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     public function runMove(dir:String, e:Event):void {
        if (dir == "up") {
           moveableButton.y = moveableButton.y - 5;
        } else if (dir == "down") {
           moveableButton.y = moveableButton.y + 5;
        } else if (dir == "left") {
           moveableButton.x = moveableButton.x - 5;
        } else if (dir == "right") {
           moveableButton.x = moveableButton.x + 5;
        }
     }
  ]]></mx:Script>

  <mx:Canvas height="100%" width="100%">
     <mx:Button id="moveableButton"
       label="{moveableButton.x.toString()},{moveableButton.y.toString()}"
        x="200"
        y="200"
        width="80"
     />
  </mx:Canvas>

  <mx:VBox horizontalAlign="center">
     <mx:Button id="b1"
        label="Up"
        click='runMove("up",event);'
        width="50"
     />
        <mx:HBox horizontalAlign="center">
           <mx:Button id="b2"
            label="Left"
            click='runMove("left",event);'
            width="50"
           />
           <mx:Button id="b3"
            label="Right"
            click='runMove("right",event);'
```

```
            width="50"
           />
        </mx:HBox>
      <mx:Button id="b4"
         label="Down"
         click='runMove("down",event);'
         width="50"
      />
   </mx:VBox>

</mx:Application>
```

Manually dispatching events

You can manually dispatch events using a component instance's `dispatchEvent()` method. All components that extend UIComponent have this method. The method is inherited from the EventDispatcher class which UIComponent extends.

The syntax for the `dispatchEvent()` method is as follows:

```
objectInstance.dispatchEvent(event:Event):Boolean
```

When dispatching an event, you must create a new Event object. The syntax for the Event object constructor is as follows:

```
Event(event_type:String, bubbles:Boolean, cancelable:Boolean)
```

The *`event_type`* parameter is the `type` property of the Event object. The *`bubbles`* and *`cancelable`* parameters are optional and both default to `false`. For information on bubbling and capturing, see "Event propagation" on page 112.

You can use the `dispatchEvent()` method to dispatch any event you want, not just a custom event. You can dispatch a Button control's `click` event, even though the user did not click a Button control, as in the following example:

```
<?xml version="1.0"?>
<!-- events/DispatchEventExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="createListener(event)">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     private function createListener(e:Event):void {
        b1.addEventListener(MouseEvent.MOUSE_OVER, myEventHandler);
        b1.addEventListener(MouseEvent.CLICK, myClickHandler);
     }

     private function myEventHandler(e:Event):void {
        var result:Boolean = b1.dispatchEvent(new
MouseEvent(MouseEvent.CLICK, true, false));
     }

     private function myClickHandler(e:Event):void {
        Alert.show("Triggered by the " + e.type + " event");
     }
  ]]></mx:Script>
  <mx:Button id="b1" label="Click Me"/>
</mx:Application>
```

You can also manually dispatch an event in an MXML tag. In the following example, moving the mouse pointer over the button triggers the button's `click` event:

```
<?xml version="1.0"?>
<!-- events/DispatchEventExampleInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="createListener(event)">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     private function createListener(e:Event):void {
        b1.addEventListener(MouseEvent.CLICK, myClickHandler);
     }

     private function myClickHandler(e:Event):void {
        Alert.show("Triggered by the " + e.type + " event");
     }
  ]]></mx:Script>
  <mx:Button id="b1" label="Click Me" mouseOver="b1.dispatchEvent(new
MouseEvent(MouseEvent.CLICK, true, false))"/>
</mx:Application>
```

Your Flex application is not required to handle the newly dispatched event. If you trigger an event that has no listeners, Flex ignores the event.

You can set properties of the Event object in ActionScript, but you cannot add new properties because the object is not dynamic. The following example intercepts a click event. It then creates a new MouseEvent object and dispatches it. In addition, it sets the value of the `shiftKey` property of the MouseEvent object to `true`, to simulate a Shift-click on the keyboard.

```
<?xml version="1.0"?>
<!-- events/DispatchCustomizedEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="addListeners()">
  <mx:Script><![CDATA[
     private function customLogEvent(e:MouseEvent):void {
        ta1.text = e.currentTarget.id + ":" +  e.type + ":" + e.shiftKey;

        // Remove current listener to avoid recursion.
        e.currentTarget.removeEventListener("click",customLogEvent);
     }

     private function handleEvent(e:MouseEvent):void {
        // Add new handler for custom event about to be dispatched.
        e.currentTarget.addEventListener("click",customLogEvent);

        // Create new event object.
        var mev:MouseEvent = new MouseEvent("click",false,false);

        // Customize event object.
        mev.shiftKey = true;

        // Dispatch custom event.
        e.currentTarget.dispatchEvent(mev);
     }

     private function addListeners():void {
        b1.addEventListener("click",handleEvent);
        b2.addEventListener("click",handleEvent);
     }

  ]]></mx:Script>

  <mx:VBox id="vb1">
     <mx:Button id="b1" label="B1"/>
     <mx:Button id="b2" label="B2"/>
     <mx:TextArea id="ta1"/>
  </mx:VBox>

</mx:Application>
```

If you want to add custom properties to an Event object, you must extend the Event object and define the new properties in your own custom class. You can then manually dispatch your custom events with the `dispatchEvent()` method, as you would any event.

If you create a custom ActionScript class that dispatches its own events but does not extend UIComponent, you can extend the flash.events.EventDispatcher class to get access to the `addEventListener()`, `removeEventListener()`, and `dispatchEvent()` methods.

For more information on creating custom classes, see *Creating and Extending Flex 2 Components*.

Event propagation

When events are triggered, there are three phases in which Flex checks whether there are event listeners. These phases occur in the following order:

- First, capturing
- Next, targeting
- Finally, bubbling

During each of these phases, the nodes have a chance to react to the event. For example, assume the user clicks a Button control that is inside a VBox container. During the capturing phase, Flex checks the Application object and the VBox for listeners to handle the event. Flex then triggers the Button's listeners in the target phase. In the bubbling phase, the VBox and then the Application are again given a chance to handle the event now in the reverse order from the order in which they were checked in the capturing phase.

In ActionScript 3.0, you can register event listeners on a target node and on any node along the event flow. Not all events, however, participate in all three phases of the event flow. Some types of events are dispatched directly to the target node and participate in neither the capturing nor the bubbling phases. All events can be captured unless they are dispatched from the top node.

Other events may target objects that are not on the display list, such as events dispatched to an instance of the Socket class. These event objects flow directly to the target node, without participating in the capturing or bubbling phases. You can also cancel an event as it flows through the event model so that even though it was supposed to continue to the other phases, you stopped it from doing so. You can do this only if the `cancelable` property is set to `true`.

Capturing and bubbling happen as the Event object moves from node to node in the display list: parent-to-child for capturing and child-to-parent for bubbling. This process has nothing to do with the inheritance hierarchy. Only DisplayObject objects (visual objects such as containers and controls) can have a capturing phase and a bubbling phase in addition to the targeting phase.

Mouse events and keyboard events are among those that bubble. Any event can be captured, but no DisplayObject objects listen during the capturing phase unless you explicitly instruct them to do so. In other words, capturing is disabled by default.

When a faceless event dispatcher, such as a Validator, dispatches an event, there is only a targeting phase, because there is no visual display list for the Event object to capture or bubble through.

About the target and currentTarget properties

Every Event object has a `target` and a `currentTarget` property that help you to keep track of where it is in the process of propagation. The `target` property refers to the dispatcher of the event. The `currentTarget` property refers to the current node that is being examined for event listeners.

When you handle a mouse event such as MouseEvent.CLICK by writing a listener on some component, the `event.target` property does not necessarily refer to that component; it is often a subcomponent, such as the Button control's UITextField, that defines the label.

When Flash Player dispatches an event, it dispatches the event from the frontmost object under the mouse. Because children are in front of parents, that means it might dispatch the event from an internal subcomponent, such as the UITextField of a Button.

The `event.target` property is set to the object that dispatched the event (in this case, UITextField), not the object that is being listened to (in most cases, you have a Button control listen for a `click` event).

MouseEvent events bubble up the parent chain, and can be handled on any ancestor. As the event bubbles, the value of the `event.target` property stays the same (UITextField), but the value of the `event.currentTarget` property is set at each level to be the ancestor that is handling the event. Eventually, the `currentTarget` will be Button, at which time the Button control's event listener will handle the event. For this reason, you should use the `event.currentTarget` property rather than the `event.target` property; for example:

```
<mx:Button label="OK" click="trace(event.currentTarget.label)"/>
```

In this case, in the Button event's click event listener, the `event.currentTarget` property always refers to the Button, while `event.target` might be either the Button or its UITextField, depending on where on the Button control the user clicked.

Capturing phase

In the capturing phase, Flex examines an event's ancestors in the display list to see if which ones are registered as a listener for the event. Flex starts with the root ancestor and continues down the display list to the direct ancestor of the target. In most cases, the root ancestors are the stage, then the SystemManager, then the Application object.

For example, if you have an application with a Panel container that contains a TitleWindow container, which in turn contains a Button control, the structure appears as follows:

```
Application
  Panel
    TitleWindow
      Button
```

If your listener is on the `click` event of the Button control, the following steps occur during the capturing phase if capturing is enabled:

1. Check the Application container for click event listeners.
2. Check the Panel container for click event listeners.
3. Check the TitleWindow container for click event listeners.

During the capturing phase, Flex changes the value of the `currentTarget` property on the Event object to match the current node whose listener is being called. The `target` property continues to refer to the dispatcher of the event.

By default, no container listens during the capturing phase. The default value of the *`use_capture`* argument is `false`. The only way to add a listener during this phase is to pass `true` for the *`use_capture`* argument when calling the `addEventListener()` method, as the following example shows:

```
myPanel.addEventListener(MouseEvent.MOUSE_DOWN, clickHandler, true);
```

If you add an event listener inline with MXML, Flex sets this argument to `false`; you cannot override it.

If you set the *`use_capture`* argument to `true`—in other words, if an event is propagated through the capturing phase—the event can still bubble, but capture phase listeners will not react to it. If you want your event to traverse both the capturing and bubbling phases, you must call `addEventListener()` twice: once with *`use_capture`* set to `true`, and then again with *`use_capture`* set to `false`.

The capturing phase is very rarely used, and it can also be computationally intensive. By contrast, bubbling is much more common.

Targeting phase

In the targeting phase, Flex invokes the event dispatcher's listeners. No other nodes on the display list are examined for event listeners. The values of the `currentTarget` and the `target` properties on the Event object during the targeting phase are the same.

Bubbling phase

In the bubbling phase, Flex examines an event's ancestors for event listeners. Flex starts with the dispatcher's immediate ancestor and continues up the display list to the root ancestor. This is the reverse of the capturing phase.

For example, if you have an application with a Panel container that contains a TitleWindow container that contains a Button control, the structure appears as follows:

```
Application
  Panel
    TitleWindow
      Button
```

If your listener is on the `click` event of the Button control, the following steps occur during the bubble phase if bubbling is enabled:

1. Check the TitleWindow container for click event listeners.
2. Check the Panel container for click event listeners.
3. Check the Application container for click event listeners.

An event only bubbles if its `bubbles` property is set to `true`. Mouse events and keyboard events are among those that bubble; it is less common for higher-level events that are dispatched by Flex to bubble. Events that can be bubbled include `change`, `click`, `doubleClick`, `keyDown`, `keyUp`, `mouseDown`, and `mouseUp`. To determine whether an event bubbles, see the event's entry in *Adobe Flex 2 Language Reference*.

During the bubbling phase, Flex changes the value of the `currentTarget` property on the Event object to match the current node whose listener is being called. The `target` property continues to refer to the dispatcher of the event.

When Flex invokes an event listener, the Event object might have actually been dispatched by an object deeper in the display list. The object that originally dispatched the event is the `target`. The object that the event is currently bubbling through is the `currentTarget`. So, you should generally use the `currentTarget` property instead of the `target` property when referring to the current object in your event listeners.

You can only register an event listener with an object if that object dispatches the event. For example, you cannot register a Form container to listen for a `click` event, even though that container contains a Button control. Form containers do not dispatch `click` events. Form containers *do* dispatch the `mouseDown` event, so you could put a mouseDown event listener on the Form container tag. If you do that, your event listener is triggered whenever the Button control or Form container receives a `mouseDown` event.

If you set the `useCapture` property to `true`—in other words, if an event is propagated through the capturing phase—then it does not bubble, regardless of its default bubbling behavior. If you want your event to traverse both the capturing and bubbling phases, you must call `addEventListener()` twice: once with `useCapture` set to `true`, and then again with `useCapture` set to `false`.

An event only bubbles up the parent's chain of ancestors in the display list. Siblings, such as two Button controls inside the same container, do not intercept each other's events.

Detecting the event phase

You can determine what phase you are in by using the Event object's `eventPhase` property. This property contains an integer that represents one of the following constants:

- 1 — Capturing phase (`CAPTURING_PHASE`)
- 2 — Targeting phase (`AT_TARGET`)
- 3 — Bubbling phase (`BUBBLING_PHASE`)

The following example displays the current phase and current target's ID:

```
<?xml version="1.0"?>
<!-- events/DisplayCurrentTargetInfo.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     private function showInfo(e:MouseEvent):void {
        trace(e.eventPhase + ":" + e.currentTarget.id);
     }
  ]]></mx:Script>

     <mx:Button id="b1" label="Click Me" click="showInfo(event)"/>

</mx:Application>
```

Stopping propagation

During any phase, you can stop the traversal of the display list by calling one of the following methods on the Event object:

- `stopPropagation()`
- `stopImmediatePropagation()`

You can call either the event's `stopPropagation()` method or the `stopImmediatePropagation()` method to prevent an Event object from continuing on its way through the event flow. The two methods are nearly identical and differ only in whether the current node's remaining event listeners are allowed to execute. The `stopPropagation()` method prevents the Event object from moving on to the next node, but only after any other event listeners on the current node are allowed to execute.

The `stopImmediatePropagation()` method also prevents the Event objects from moving on to the next node, but it does not allow any other event listeners on the current node to execute.

The following example creates a TitleWindow container inside a Panel container. Both containers are registered to listen for a `mouseDown` event. As a result, if you click on the TitleWindow container, the `showAlert()` method is called twice unless you add a call to the `stopImmediatePropagation()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/StoppingPropagation.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="init(event)">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     import flash.events.MouseEvent;
     import flash.events.Event;

     public function init(e:Event):void {
        p1.addEventListener(MouseEvent.MOUSE_DOWN,showAlert);
        tw1.addEventListener(MouseEvent.MOUSE_DOWN,showAlert);
        tw1.addEventListener(Event.CLOSE,closeWindow);
     }
     public function showAlert(e:Event):void {
        Alert.show("Alert!\n" + e.currentTarget + "\n" + e.eventPhase);
        e.stopImmediatePropagation();
     }
     public function closeWindow(e:Event):void {
        p1.removeChild(tw1);
     }
  ]]></mx:Script>

  <mx:Panel id="p1" title="Panel 1">
     <mx:TitleWindow id="tw1" width="300" height="300"
showCloseButton="true" title="Title Window 1">
        <mx:Button label="Enter name"/>
        <mx:TextArea id="ta1"/>
     </mx:TitleWindow>
  </mx:Panel>
</mx:Application>
```

Examples

In the following example, the parent container's click handler disables the target control after the target handles the event. It shows that you can reuse the logic of a single listener (click on the HBox container) for multiple events (all the clicks).

```
<?xml version="1.0"?>
<!-- events/NestedHandlers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     public function disableControl(event:MouseEvent):void {
        event.currentTarget.enabled = false;
     }
     public function doSomething(event:MouseEvent):void {
        b1.label = "clicked";
        ta1.text += "something wonderful happened";
     }
     public function doSomethingElse(event:MouseEvent):void {
        b2.label = "clicked";
        ta1.text += "something wonderful happened again";
     }
  ]]></mx:Script>
  <mx:HBox height="50" click="disableControl(event)">
     <mx:Button id='b1' label="Click Me" click="doSomething(event)"/>
     <mx:Button id='b2' label="Click Me" click="doSomethingElse(event)"/>
     <mx:TextArea id="ta1"/>
  </mx:HBox>
</mx:Application>
```

By having a single listener on a parent control instead of many listeners (one on each child control), you can reduce your code size and make your applications more efficient. Reducing the amount of calls to the `addEventListener()` method potentially reduces application startup time and memory usage.

The following example registers an event handler for the Panel container, rather than registering a listener for each link. All children of the Panel container inherit this event handler. Since Flex invokes the handler on a bubbled event, you use the `target` property rather than the `currentTarget` property. In this handler, the `currentTarget` property would refer to the Panel control, whereas the `target` property refers to the LinkButton control, which has the label that you want.

```
<?xml version="1.0"?>
<!-- events/SingleRegisterHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="createLinkHandler()">
  <mx:Script><![CDATA[
  private function linkHandler(event:MouseEvent):void {
     var url:URLRequest = new URLRequest("http://finance.google.com/
finance?q=" + event.target.label);
     navigateToURL(url);
  }
  private function createLinkHandler():void {
     p1.addEventListener(MouseEvent.CLICK,linkHandler);
  }
  ]]></mx:Script>
  <mx:Panel id="p1" title="Click on a stock ticker symbol">
     <mx:LinkButton label="ADBE"/>
     <mx:LinkButton label="GE"/>
     <mx:LinkButton label="IBM"/>
     <mx:LinkButton label="INTC"/>
  </mx:Panel>
</mx:Application>
```

Event priorities

You can register any number of event listeners with a single event. Flex registers event listeners in the order in which the `addEventListener()` methods are called. Flex then calls the listener functions when the event occurs in the order in which they were registered. However, if you register some event listeners inline and some with the `addEventListener()` method, the order in which the listeners are called for a single event can be unpredictable.

You can change the order in which Flex calls event listeners by using the *`priority`* parameter of the `addEventListener()` method. It is the fourth argument of the `addEventListener()` method.

Flex calls event listeners in priority order, from highest to lowest. The highest priority event is called first. In the following example, Flex calls the `verifyInputData()` method before the `saveInputData()` function. The `verifyInputData()` method has the highest priority. The last method to be called is `returnResult()` because the value of its `priority` parameter is lowest.

```
<?xml version="1.0"?>
<!-- events/ShowEventPriorities.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[

  private function returnResult(e:Event):void {
     trace("returnResult");
  }

  private function verifyInputData(e:Event):void {
     trace("verifyInputData");
  }

  private function saveInputData(e:Event):void {
     trace("saveInputData");
  }

  private function initApp():void {
     b1.addEventListener(MouseEvent.CLICK, returnResult, false, 1);
     b1.addEventListener(MouseEvent.CLICK, saveInputData, false, 2);
     b1.addEventListener(MouseEvent.CLICK, verifyInputData, false, 3);
  }
  ]]></mx:Script>

  <mx:Button id="b1" label="Click Me"/>

</mx:Application>
```

You can set the event priority to any valid integer, positive or negative. The default value is 0. If multiple listeners have the same priority, Flex calls them in the order in which they were registered.

If you want to change the priority of an event listener once the event listener has already been defined, you must remove the listener by calling the `removeEventListener()` method. You add the event again with the new priority.

The *`priority`* parameter of the `addEventListener()` method is not an official part of the DOM Level 3 events model. ActionScript 3.0 provides it so that programmers can be more flexible when organizing their event listeners.

Even if you give a listener a higher priority than other listeners, there is no way to guarantee that the listener will finish executing before the next listener is called. You should ensure that listeners do not rely on other listeners completing execution before calling the next listener. It is important to understand that Flash Player does not necessarily wait until the first event listener finishes processing before proceeding with the next one.

If your listeners do rely on each other, you can call one listener function from within another, or dispatch a new event from within the first event listener. For more information on manually dispatching events, see "Manually dispatching events" on page 109.

Using event subclasses

Depending on the event type, the Event object can have a wide range of properties. These properties are based on those defined in the W3C specification (www.w3.org/TR/DOM-Level-3-Events/events.html), but Flex does not implement all of these.

When you declare an Event object in a listener function, you can declare it of type Event, or you can specify a subclass of the Event object. In the following example, you specify the event object as type MouseEvent:

```
public function performAction(e:MouseEvent):void {
   ...
}
```

Most controls generate an object that is of a specific event type; for example, a mouse click generates an object of type MouseEvent. By specifying a more specific event type, you can access specific properties without having to cast the Event object to something else. In addition, some subclasses of the Event object have methods that are unique to them. For example, the LogEvent has a `getLevelString()` method, which returns the log level as a String. The generic Event object does not have this method.

An event object that you define at run time can be a subclass of the compile-time type. You can access the event-specific properties inside an event listener even if you did not declare the specific event type, as long as you cast the Event object to a specific type. In the following example, the function defines the object type Event. However, inside the function, in order to access the `localX` and `localY` properties, which are specific to the MouseEvent class, you must cast the Event object to be of type MouseEvent.

```
<?xml version="1.0"?>
<!-- events/AccessEventSpecificProperties.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="addListeners()">
  <mx:Script><![CDATA[
     private function customLogEvent(e:Event):void {
        var a:MouseEvent = MouseEvent(e);
        trace(a.localY + ":" + a.localX);
     }

     private function addListeners():void {
        b1.addEventListener(MouseEvent.CLICK, customLogEvent);
     }
  ]]></mx:Script>
  <mx:VBox id="vb1">
     <mx:Button id="b1" label="Click Me"/>
  </mx:VBox>
</mx:Application>
```

If you declare the Event object as a specific type, you are not required to cast that object in the handler, as the following example shows:

```
private function customLogEvent(e:MouseEvent):void { ... }
```

In the previous example, you can also cast the Event object for only the property access, using the syntax shown in the following example:

```
<?xml version="1.0"?>
<!-- events/SinglePropertyAccess.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="addListeners()">
  <mx:Script><![CDATA[
     private function customLogEvent(e:Event):void {
        trace(MouseEvent(e).localY + ":" + MouseEvent(e).localX);
     }

     private function addListeners():void {
        b1.addEventListener(MouseEvent.CLICK, customLogEvent);
     }
  ]]></mx:Script>
  <mx:VBox id="vb1">
     <mx:Button id="b1" label="Click Me"/>
  </mx:VBox>
</mx:Application>
```

This approach can use less memory and system resources, but it is best to declare the event's type as specifically as possible.

Each of the Event object's subclasses provides additional properties and event types that are unique to that category of events. The MouseEvent class defines several event types related to that input device, including the `CLICK`, `DOUBLE_CLICK`, `MOUSE_DOWN`, and `MOUSE_UP` event types.

For a list of types for each Event subclass, see the subclass's entry in *Adobe Flex 2 Language Reference*.

About keyboard events

It is common for applications to respond to a key or series of keys and perform some action—for example, Control+q to quit the application. While Flash Player supports all the basic functionality of key combinations from the underlying operating system, it also lets you override or trap any key or combination of keys to perform a custom action.

Handling keyboard events

In some cases, you want to trap keys globally, meaning no matter where the user is in the application, their keystrokes are recognized by the application and the action is performed. Flex recognizes global keyboard events whether the user is hovering over a button or the focus is inside a TextInput control.

A common way to handle global key presses is to create a listener for the KeyboardEvent.KEY_DOWN or KeyboardEvent.KEY_UP event on the application. Listeners on the application container are triggered every time a key is pressed, regardless of where the focus is. Inside the handler, you can examine the key code or the character code using the `charCode` and `keyCode` properties of the KeyboardEvent class, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/TrapAllKeys.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     private function initApp():void {
        application.addEventListener(KeyboardEvent.KEY_UP,keyHandler);

        // Set the focus somewhere inside the application.
        myCanvas.setFocus();
     }

     // Quit the application by closing the browser using JavaScript
     // if the user presses Shift+Q. This may not work in all browsers.
     private function keyHandler(event:KeyboardEvent):void {
           trace(event.keyCode + "/" + event.charCode);
          var url:URLRequest = new URLRequest("javascript:window.close()");
           navigateToURL(url,"_self");
     }
  ]]></mx:Script>

  <mx:Canvas id="myCanvas"/>

</mx:Application>
```

Because any class that extends UIComponent dispatches the `keyUp` and `keyDown` events, you can also trap keys pressed when the focus is on an individual component, as in the following example:

```
<?xml version="1.0"?>
<!-- events/TrapKeysOnTextArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     private function keyHandler(event:KeyboardEvent):void {
           trace(event.keyCode + "/" + event.charCode);
     }
  ]]></mx:Script>

  <mx:TextInput id="my_input" keyUp="keyHandler(event)"/>

</mx:Application>
```

Understanding the keyCode and charCode properties

You can access the `keyCode` and `charCode` properties to determine what key was pressed and trigger other actions as a result. The `keyCode` property is a numeric value that corresponds to the value of a key on the keyboard. The `charCode` property is the numeric value of that key in the current character set (the default character set is UTF-8, which supports ASCII). The primary difference between the key code and character values is that a key code value represents a particular key on the keyboard (the 1 on a keypad is different than the 1 in the top row, but the 1 on the keyboard and the key that generates the ! are the same key), and the character value represents a particular character (the *R* and *r* characters are different).

The mappings between keys and key codes is device and operating system dependent. ASCII values, on the other hand, are available in the ActionScript documentation.

The following example shows the character and key code values for the keys you press. When you run this example, you must be sure to put the focus in the application before beginning.

```
<?xml version="1.0"?>
<!-- charts/ShowCharAndKeyCodes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import flash.events.KeyboardEvent;

     private function init():void {
        ti1.setFocus();
        this.addEventListener(KeyboardEvent.KEY_DOWN, trapKeys);
     }

     private function trapKeys(e:KeyboardEvent):void {
        l0.text = String(e.toString());
```

```
            l1.text = numToChar(e.charCode) + " (" + String(e.charCode) + ")";
            l2.text = numToChar(e.keyCode) + " (" + String(e.keyCode) + ")";
        }

        private function numToChar(num:int):String {
            if (num > 47 && num < 58) {
                var strNums:String = "0123456789";
                return strNums.charAt(num - 48);
            } else if (num > 64 && num < 91) {
                var strCaps:String = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
                return strCaps.charAt(num - 65);
            } else if (num > 96 && num < 123) {
                var strLow:String = "abcdefghijklmnopqrstuvwxyz";
                return strLow.charAt(num - 97);
            } else {
                return num.toString();
            }
        }
    ]]></mx:Script>

    <mx:TextInput width="50%" id="ti1"/>

    <mx:Canvas id="mainCanvas" width="100%" height="100%">
        <mx:Form>
            <mx:FormItem label="Char (Code)">
                <mx:Label id="l1"/>
            </mx:FormItem>
            <mx:FormItem label="Key (Code)">
                <mx:Label id="l2"/>
            </mx:FormItem>
            <mx:FormItem label="Key Event">
                <mx:Label id="l0"/>
            </mx:FormItem>
        </mx:Form>
    </mx:Canvas>

</mx:Application>
```

You can listen for specific keys or combinations of keys by using a conditional operator in the KeyboardEvent handler. The following example listens for the combination of the Shift key plus the q key and prompts the user to close the browser window if they press those keys at the same time:

```
<?xml version="1.0"?>
<!-- events/TrapQKey.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     private function initApp():void {
        application.addEventListener(KeyboardEvent.KEY_UP,keyHandler);
        // Set the focus somewhere inside the application.
        myCanvas.setFocus();
     }

     //This function quits the application if the user presses Shift+Q.
     private function keyHandler(event:KeyboardEvent):void {
        var bShiftPressed:Boolean = event.shiftKey;
        if (bShiftPressed) {
           var curKeyCode:int = event.keyCode;
           if (curKeyCode == 81) { // 81 is the keycode value for the Q key
            // Quit the application by closing the browser using JavaScript.
             // This may not work in all browsers.
             var url:URLRequest = new
             URLRequest("javascript:window.close()");
             navigateToURL(url,"_self");
           }
        }
     }
  ]]></mx:Script>

  <mx:Canvas id="myCanvas"/>

</mx:Application>
```

Notice that this application must have focus when you run it in a browser so that the application can capture keyboard events.

Understanding KeyboardEvent precedence

If you define `keyUp` or `keyDown` event listeners for both a control and its parent, you will notice that the keyboard event is dispatched for each component because the event bubbles. The only difference is that the `currentTarget` property of the KeyboardEvent object is changed.

In the following example, the application, the `my_vbox` container, and the `my_textinput` control all dispatch `keyUp` events to the `keyHandler()` event listener function:

```
<?xml version="1.0"?>
<!-- events/KeyboardEventPrecedence.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     private function initApp():void {
        application.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
        my_vbox.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
        my_textinput.addEventListener(KeyboardEvent.KEY_UP, keyHandler);

        // Set the focus somewhere inside the application.
        my_textinput.setFocus();
     }

     private function keyHandler(event:KeyboardEvent):void {
           trace(event.target + "(" + event.currentTarget + "): " +
event.keyCode + "/" + event.charCode);
     }
  ]]></mx:Script>

  <mx:VBox id="my_vbox">
     <mx:TextInput id="my_textinput"/>
  </mx:VBox>

</mx:Application>
```

When you examine the `trace()` method output, you will notice that the `target` property of the KeyboardEvent object stays the same because it refers to the original dispatcher of the event (in this case, my_textinput). But the `currentTarget` property changes depending on what the current node is during the bubbling (in this case, it changes from my_textinput to my_vbox to the application itself).

The order of calls to the event listener is determined by the object hierarchy and not the order in which the `addEventListener()` methods were called. Child controls dispatch events before their parents. In this example, for each key pressed, the TextInput control dispatches the event first, the VBox container next, and finally the application.

When handling a key or key combination that the underlying operating system or browser recognizes, the operating system or browser generally processes the event first. For example, in Microsoft Internet Explorer, pressing Control+w closes the browser window. If you trap that combination in your Flex application, Internet Explorer users never know it, because the browser closes before the ActiveX Flash Player has a chance to react to the event.

Handling keyboard-related MouseEvents

The MouseEvent class and all MouseEvent subclasses (such as ChartItemEvent, DragEvent, ItemClickEvent, and LegendMouseEvent) have the following properties that you can use to determine if a specific key was held down when the event occurred:

Property	Description
`altKey`	Is set to `true` if the Alt key was held down when the user pressed the mouse button; otherwise, `false`.
`ctrlKey`	Is set to `true` if the Control key was held down when the user pressed mouse button; otherwise, `false`.
`shiftKey`	Is set to `true` if the Shift key was held down when the user pressed mouse button; otherwise, `false`.

The following example deletes button controls, based on whether the user holds down the Shift key while pressing the mouse button:

```
<?xml version="1.0"?>
<!-- events/DetectingShiftClicks.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.controls.Button;

     private function initApp():void {
        var b1:Button = new Button();
        var b2:Button = new Button();

        b1.addEventListener(MouseEvent.CLICK, removeButtons);
        b2.addEventListener(MouseEvent.CLICK, removeButtons);

        addChild(b1);
        addChild(b2);
     }

     private function removeButtons(event:MouseEvent):void {
        if (event.shiftKey) {
           removeChild(Button(event.currentTarget));
        } else {
          event.currentTarget.toolTip = "You must hold the shiftkey down to
delete this button.";
        }
     }
  ]]></mx:Script>

</mx:Application>
```

PART 2

Building User Interfaces for Flex Applications

2

This part describes how to use Adobe Flex 2 components to build the user interface for your application.

The following topics are included:

CHAPTER 6

Using Flex Visual Components

6

You use visual components to build Adobe Flex applications. Visual components (often referred to as components for brevity) have a flexible set of characteristics that let you control and configure them as necessary to meet your application's requirements. This topic provides an overview of components, component syntax, and component configuration.

Contents

About visual components

Flex includes a component-based development model that you use to develop your application and its user interface. You can use the prebuilt visual components included with Flex, you can extend components to add new properties and methods, and you can create components as required by your application.

Visual components are extremely flexible and provide you with a great deal of control over the component's appearance, how the component responds to user interactions, the font and size of any text included in the component, the size of the component in the application, and many other characteristics.

This topic contains an overview of many of the characteristics of visual components, including the following:

Size Height and width of a component. All visual components have a default size. You can use the default size, specify your own size, or let Flex resize a component as part of laying out your application.

Events Application or user actions that require a component response. Events include component creation, mouse actions such as moving the mouse over a component, and button clicks.

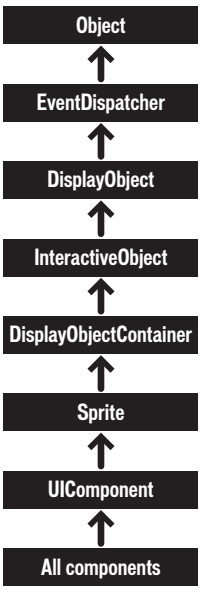
Styles Characteristics such as font, font size, and text alignment. These are the same styles that you define and use with Cascading Style Sheets (CSS).

Behaviors Visible or audible changes to the component triggered by an application or user action. Examples of behaviors are moving or resizing a component based on a mouse click.

Skins Classes that control a visual component's appearance.

Class hierarchy for visual components

Flex visual components are implemented as a class hierarchy in ActionScript. Therefore, each visual component in your application is an instance of an ActionScript class. The following image shows this hierarchy in detail up to the Flash Sprite component level:



All visual components are derived from the UIComponent class and its superclasses, the Flash Sprite through Object classes, and inherit the properties and methods of their superclasses. In addition, visual components inherit other characteristics of the superclasses, including event, style, and behavior definitions.

Using the UIComponent class

The UIComponent class is the base class for all Flex visual components. For detailed documentation, see UIComponent in *Adobe Flex 2 Language Reference*.

Commonly used UIComponent properties

The following table lists only the most commonly used properties of components that extend the UIComponent class:

Property	Type	Description
`doubleClickEnabled`	Boolean	Setting to `true` lets the component dispatch a `doubleClickEvent` when a user presses and releases the mouse button twice in rapid succession over the component.
`enabled`	Boolean	Setting to `true` lets the component accept keyboard focus and mouse input. The default value is `true`. If you set `enabled` to `false` for a container, Flex dims the color of the container and all of its children, and blocks user input to the container and to all its children.
`height`	Number	The height of the component, in pixels. In MXML tags, but not in ActionScript, you can set this property as a percentage of available space by specifying a value such as 70%; in ActionScript, you must use the `percentHeight` property. The property always returns a number of pixels. In ActionScript, you use the `perCent`
`id`	String	Specifies the component identifier. This value identifies the specific instance of the object and should not contain any white space or special characters. Each component in a Flex document must have a unique `id` value. For example, if you have two custom components, each component can include one, and only one Button control with the id "okButton".

Property	Type	Description
percentHeight	Number	The height of the component as a percentage of its parent container, or for <mx:Application> tags, the full height of the browser. Returns NaN if a percent-based width has never been set or if a width property was set after the percentWidth was set.
percentWidth	Number	The width of the component as a percentage of its parent container, or for <mx:Application> tags, the full span of the browser. Returns NaN if a percent-based width has never been set or if a width property was set after the percentWidth was set.
styleName	String	Specifies the style class selector to apply to the component.
toolTip	String	Specifies the text string displayed when the mouse pointer hovers over that component.
visible	Boolean	Specifies whether the container is visible or invisible. The default value is true, which specifies that the container is visible.
width	Number	The width of the component, in pixels. In MXML tags, but not in ActionScript, you can set this property as a percentage of available space by specifying a value such as 70%; in ActionScript, you must use the percentWidth property. The property always returns a number of pixels.
x	Number	The component's *x* position within its parent container. Setting this property directly has an effect only if the parent container uses absolute positioning.
y	Number	The component's *y* position within its parent container. Setting this property directly has an effect only if the parent container uses absolute positioning.

Using components in MXML and ActionScript

Every Flex component has an MXML API and an ActionScript API. A component's MXML tag attributes are equivalent to its ActionScript properties, styles, behaviors, and events. You can use both MXML and ActionScript when working with components.

To configure a component:

1. Set the value of a component property, event, style, or behavior declaratively in an MXML tag, or at run time in ActionScript code.
2. Call a component's methods at run time in ActionScript code. The methods of an ActionScript class are not exposed in the MXML API.

The following example creates a Button control in MXML. When the user clicks the Button control, it updates the text of a TextArea control by using an ActionScript function.

```
<?xml version="1.0"?>
<!-- components\ButtonApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function handleClick():void {
                text1.text="Thanks for the click!";
            }
        ]]>
    </mx:Script>

    <mx:Button id="button1"
        label="Click here!"
        width="100"
        fontSize="12"
        click="handleClick();"/>
    <mx:TextArea id="text1"/>
</mx:Application>
```

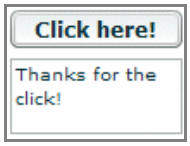
This example has the following elements:

- The `id` property is inherited by the Button control from the UIComponent class. You use it to specify an identifier for the component. This property is optional, but you must specify it if you want to access the component in ActionScript.
- The `label` property is defined by the Button control. It specifies the text that appears in the button.
- The `width` property is inherited from the UIComponent class. It optionally specifies the width of the button, in pixels.
- The Button control dispatches a `click` event when the user when a user presses and releases the main mouse button. The MXML `click` attribute specifies the ActionScript code to execute in response to the event.
- The `fontSize` style is inherited from the UIComponent class. It specifies the font size of the label text, in pixels.

The `click` event attribute can also take ActionScript code directly as its value, without your having to specify it in a function. Therefore, you can rewrite this example as the following code shows:

```
<?xml version="1.0"?>
<!-- components\ButtonAppAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Button id="button1"
        label="Click here!"
        width="100"
        fontSize="12"
        click="text1.text='Thanks for the click!';"/>

    <mx:TextArea id="text1"/>
</mx:Application>
```

Both of these examples result in the following image, shown after the button was clicked:



NOTE Although you can specify multiple lines of ActionScript code, separated by semicolons, as the value of the `click` event attribute, for readability you should limit the `click` event to only one or two lines of code.

Initializing components at run time

Flex uses MXML property attributes to initialize your components. However, you might want to use some logic to determine initial values at run time. For example, you might want to initialize a component with the current date or time. Flex must calculate this type of information when the application executes.

Every component dispatches several events during its life cycle. In particular, all components dispatch the following events that let you specify ActionScript to initialize a component:

preInitialize Dispatched when a component has been created in a rough state, and any children have not been created.

initialize Dispatched when a component and all its children have been created, but before the component size has been determined.

creationComplete Dispatched when the component has been laid out and the component is visible (if appropriate).

NOTE

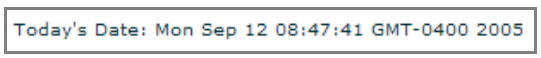
For more information on how components are created, see "About the component instantiation life cycle" on page 149.

You can use the `initialize` event to configure most component characteristics; in particular, use it to configure any value that affects the component's size. Use the `creationComplete` event if your initialization code must get information about the component layout.

The following example configures Flex to call the `initDate()` function when it initializes the Label control. When Flex finishes initializing the Label control, and before the application appears, Flex calls the `initDate()` function.

```
<?xml version="1.0"?>
<!-- components\LabelInit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function initDate():void {
                label1.text += new Date();
            }
        ]]>
    </mx:Script>

    <mx:Box borderStyle="solid">
        <mx:Label id="label1"
            text="Today's Date: "
            initialize="initDate();"/>
    </mx:Box>
</mx:Application>
```

This example produces the following image:



You can also express the previous example without a function call by adding the ActionScript code in the component's definition. The following example does the same thing, but without an explicit function call:

```
<?xml version="1.0"?>
<!-- components\LabelInitAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Box borderStyle="solid">
        <mx:Label id="label1"
            text="Today's Date:"
            initialize="label1.text += new Date();"/>
    </mx:Box>
</mx:Application>
```

As with other calls that are embedded within component definitions, you can add multiple ActionScript statements to the `initialize` MXML attribute by separating each function or method call with a semicolon. The following example calls the `initDate()` function and writes a message in the flexlog.txt file when the `label1` component is instantiated:

```
<?xml version="1.0"?>
<!-- components\LabelInitASAndEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function initDate():void {
                label1.text += new Date();
            }
        ]]>
    </mx:Script>

    <mx:Box borderStyle="solid">
        <mx:Label id="label1"
            text="Today's Date:"
            initialize="initDate(); trace('The label is initialized!');"/>
    </mx:Box>
</mx:Application>
```

Configuring components: syntax summary

The following table summarizes the MXML and ActionScript component APIs that you use to configure components:

	MXML example	ActionScript example
Read-write property	`<mx:Tile id="tile1" label="My Tile" visible="true"/>`	`tile1.label="My Tile";` `tile1.visible=true;`
Read-only property	You cannot use a read-only property as an attribute in MXML.	To get the value of a read-only property: `var theClass:String=mp1.className;`
Method	Methods are not available in MXML.	`myList.sortItemsBy("data", "DESC");`

Event	`<mx:Accordion id="myAcc" change="changeHandler (event);"/>`
	
	(You must also define a `changeHandler()` function as shown in the ActionScript example.
		`private function changeHandler(event:MouseEvent):void {`
		`...`
		`}`
		
		`myButton.addEventListener("click", changeHandler);`

Style	`<mx:Tile id="tile1" paddingTop="12" paddingBottom="12"/>`	To set the style:
		`tile1.setStyle("paddingTop", 12);`
		`tile1.setStyle("paddingBottom", 12);`
		To get the style:
		`var currentPaddingTop:Number = tile1.getStyle("paddingTop");.`

Behavior	`<mx:Tile id="tile1" showEffect="{AWipeEffect}"/>`	To set the behavior:
		`myButton.setStyle('showEffect', AWipeEffect);`
		To get the behavior:
		`var currentShowEffect:String = tile1.getStyle("showEffect");`

Sizing visual components

The Flex sizing and layout mechanisms provide several ways for you to control the size of visual components:

Default sizing Flex automatically determines the sizes of controls and containers. To use default sizing, you do not specify the component's dimensions or layout constraints.

Explicit sizing You set the `height` and `width` properties to pixel values. When you do this, you set the component dimension to absolute sizes, and Flex does not override these values. The following `<mx:Application>` tag, for example, sets explicit Application dimensions:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  height="300"
  width="600">
```

Percentage-based sizing You specify the component size as a percentage of its container size. To do this, you specify the `percentHeight` and `percentWidth` properties, or, in an MXML tag, set the `height` and `width` properties to percentage values such as 100%. The following code, for example, sets percentage-based dimensions for an HBox container:

```
<mx:HBox id="hBox1" xmlns:mx="http://www.adobe.com/2006/mxml"
  height="30%"
  width="90%"/>
```

The following ActionScript line resets the HBox width to a different percentage value:

```
hBox1.percentWidth=40;
```

Constraint-based layout You can control size *and* position by anchoring components sides or centers to locations in their container by specifying the `top`, `bottom`, `left`, `right`, `horizontalCenter`, and `verticalCenter` styles. You can use constraint-based layout only for the children of a container that uses absolute layout; the Application and Panel containers can optionally use this layout, and the Canvas container always uses it. The following example uses constraint-based layout to position an HBox horizontally, and explicit sizing and positioning to determine the vertical width and position:

```
<mx:HBox id="hBox2" xmlns:mx="http://www.adobe.com/2006/mxml"
  left="30"
  right="30"
  y="150"
  height="100"/>
```

You can mix sizing techniques; however, you must ensure that the mix is appropriate. Do not specify more than one type of sizing for a component dimension; for example, do not specify a `height` and a `percentHeight` for a component. Also, ensure that the resulting sizes are appropriate; for example, if you do not want scroll bars or clipped components, ensure that the sizes of a container's children do not exceed the container size.

For detailed information on how Flex sizes components, and how you can specify sizing, see Chapter 8, "Sizing and Positioning Components," on page 221.

Examples of component sizing

The following example shows sizing within an explicitly sized container when some of the container's child controls are specified with explicit widths and some with percentage-based widths. It shows the flexibility and the complexities involved in determining component sizes. The application logs the component sizes to flashlog.txt, so you can confirm the sizing behavior.

```
<?xml version="1.0"?>
<!-- components\CompSizing.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logSizes();">

    <mx:Script>
        <![CDATA[
            private function logSizes():void {
                trace("HBox: "+ hb1.width);
                trace("Label: "+ lb1.width);
                trace("Image: "+ img1.width);
                trace("Button: "+ b1.width);
            }
        ]]>
    </mx:Script>

    <mx:HBox id="hb1" width="250">
        <mx:Label id="lb1"
            text="Hello"
            width="50"/>
        <mx:Image id="img1"
            source="@Embed(source='assets/flexlogo.jpg')"
            width="75%"/>
        <mx:Button id="b1"
            label="Button"
            width="25%"/>
    </mx:HBox>
</mx:Application>
```

The application consists of a 250-pixel-wide HBox container that contains a 50-pixel-wide label, an image that requests 75% of the container width, and a button that requests 25% of the container width. The component sizes are determined as follows:

1. Flex reserves 50 pixels for the explicitly sized Label control.
2. Flex puts an 8-pixel gap between components by default, so it reserves 16 pixels for the gaps; this leaves 184 pixels available for the two percentage-based components.

3. The minimum width of the Button component, if you do not specify an explicit width, fits the label text plus padding around it. In this case, the minimum size is 65 pixels. This value is larger than 25% of the component, so Flex does not use the percentage request, and reserves 65 pixels for the Button control.
4. The percentage-based image requests 75% of 250 pixels, or 187 pixels, but the available space is only 119 pixels, which it takes.

If you change the button and image `size` properties to 50%, the minimum button size is smaller than the requested size, so the percentage-sized controls each get 50% of the available space, 92 pixels.

The following example uses explicit sizing for the Image control and default sizing for the Button control and yields the same results as the initial example:

```
<?xml version="1.0"?>
<!-- components\CompSizingExplicit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logSizes();">

    <mx:Script>
        <![CDATA[
            private function logSizes():void {
                trace("HBox: "+ hb1.width);
                trace("Label: "+ lb1.width);
                trace("Image: "+ img1.width);
                trace("Button: "+ b1.width);
            }
        ]]>
    </mx:Script>

    <mx:HBox id="hb1" width="250">
        <mx:Label id="lb1"
            text="Hello"
            width="50"/>
        <mx:Image id="img1"
            source="@Embed(source='assets/flexlogo.jpg')"
            width="119" />
        <mx:Button id="b1"
            label="Button"/>
    </mx:HBox>
</mx:Application>
```

Percentage-based sizing removes the need to explicitly consider the gaps and margins of a container in sizing calculations, but its greatest benefit applies when containers resize. Then, the percentage-based children resize automatically based on the available container size. In the following example, the series of controls on the left resize as you resize your browser, but the corresponding controls on the right remain a fixed size because their container is fixed. Click the first Button control to logs the component sizes to flashlog.txt.

```
<?xml version="1.0"?>
<!-- components\CompSizingPercent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function logSizes():void {
                trace("HBox: "+ hb1.width);
                trace("Label: "+ lb1.width);
                trace("Image: "+ img1.width);
                trace("Button: "+ b1.width);
            }
        ]]>
    </mx:Script>

    <mx:HBox width="100%">
        <mx:HBox id="hb1" width="40%" borderStyle="solid">
            <mx:Label id="lb1"
                text="Hello"
                width="50"/>
            <mx:Image id="img1"
                source="@Embed(source='assets/flexlogo.jpg')"
                width="60%"/>
            <mx:Button id="b1"
                label="Button"
                width="40%"
                click="logSizes();"/>
        </mx:HBox>

        <mx:HBox width="260" borderStyle="solid">
            <mx:Label
                text="Hello"
                width="50"/>
            <mx:Image
                source="@Embed(source='assets/flexlogo.jpg')"
                width="119" />
            <mx:Button
                label="Button"/>
        </mx:HBox>
    </mx:HBox>
</mx:Application>
```

For more information about sizing considerations, see "Sizing components" on page 228.

Handling events

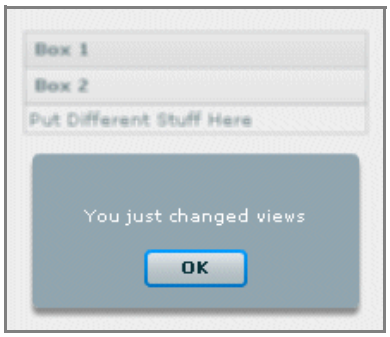
Flex applications are event-driven. *Events* let a programmer know when the user has interacted with an interface component, and also when important changes have happened in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing.

When an instance of a component dispatches an event, objects that have registered as *listeners* for that event are notified. You define event listeners, also called event *handlers*, in ActionScript to process events. You register event listeners for events either in the MXML declaration for the component or in ActionScript. For additional examples of the event handling, see "Initializing components at run time" on page 138.

The following example registers an event listener in MXML that is processed when you change views in an Accordion container.

```
<?xml version="1.0"?>
<!-- components\CompIntroEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="300"
    height="280">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private function handleAccChange():void {
                Alert.show("You just changed views");
            }
        ]]>
    </mx:Script>

    <!-- The Accordion control dispatches a change event when the
            selected child container changes. -->
    <mx:Accordion id="myAcc"
        height="60"
        width="200"
        change="handleAccChange();">
        <mx:HBox label="Box 1">
            <mx:Label text="Put Some Stuff Here"/>
        </mx:HBox>

        <mx:HBox label="Box 2">
            <mx:Label text="Put Different Stuff Here"/>
        </mx:HBox>
    </mx:Accordion>
</mx:Application>
```

This example produces the following image:



You can pass an event object, which contains information about the event, from the component to the event listener.

For the Accordion container, the event object passed to the event listener for the `change` event is of class IndexChangedEvent. You can write your event listener to access the event object, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\CompIntroEventAcc.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="300"
    height="280">

    <mx:Script>
        <![CDATA[
            // Import the class that defines the event object.
            import mx.events.IndexChangedEvent;
            import mx.controls.Alert;

            private function handleChange(event:IndexChangedEvent):void {
                var currentIndex:int=event.newIndex;
                Alert.show("You just changed views \nThe new index is "
                    + event.newIndex);
            }
        ]]>
    </mx:Script>

    <!-- The Accordion control dispatches a change event when the
            selected child container changes. -->
    <mx:Accordion id="myAcc"
        height="60"
        width="200"
        change="handleChange(event);">
        <mx:HBox label="Box 1">
            <mx:Label text="Put Some Stuff Here"/>
        </mx:HBox>

        <mx:HBox label="Box 2">
            <mx:Label text="Put Different Stuff Here"/>
        </mx:HBox>
    </mx:Accordion>
</mx:Application>
```

In this example, you access the `newIndex` property of the IndexChangedEvent object to determine the index of the new child of the Accordion container. For more information on events, see Chapter 5, “Using Events,” on page 83.

About the component instantiation life cycle

The component instantiation life cycle describes the sequence of steps that occur when you create a component object from a component class. As part of that life cycle, Flex automatically calls component methods, dispatches events, and makes the component visible.

The following example creates a Button control and adds it to a container:

```
<?xml version="1.0"?>
<!-- components\AddButtonToContainer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Box id="box1" width="200">
        <mx:Button id="button1" label="Submit"/>
    </mx:Box>
</mx:Application>
```

The following ActionScript is equivalent to the portion of the MXML code. Flex executes the same sequence of steps in both examples.

```
// Create a Box container.
var box1:Box = new Box();
// Configure the Box container.
box1.width=200;

// Create a Button control.
var button1:Button = new Button()
// Configure the Button control.
button1.label = "Submit";

// Add the Button control to the Box container.
box1.addChild(button1);
```

The following steps show what occurs when you execute the ActionScript code to create the Button control, and add it to the Box container. When you create the component in MXML, Flex 2 SDK generates equivalent code.

1. You call the component's constructor, as the following code shows:

   ```
   // Create a Button control.
   var button1:Button = new Button()
   ```

2. You configure the component by setting its properties, as the following code shows:

   ```
   // Configure the button control.
   button1.label = "Submit";
   ```

3. You call the `addChild()` method to add the component to its parent, as the following code shows:

   ```
   // Add the Button control to the Box container.
   box1.addChild(button1);
   ```

Flex performs the following actions to process this line:

a. Flex sets the `parent` property for the component to reference its parent container.
b. Flex computes the style settings for the component.
c. Flex dispatches the `add` event from the button
d. Flex dispatches the `childAdd` event from the parent container.
e. Flex dispatches the `preinitialize` event on the component. The component is in a very raw state when this event is dispatched. Many components, such as the Button control, create internal child components to implement functionality; for example, the Button control creates an internal UITextField component to represent its label text. When Flex dispatches the `preinitialize` event, the children, including the internal children, of a component have not yet been created.
f. Flex creates and initializes the component's children, including the component's internal children.
g. Flex dispatches the `initialize` event on the component. At this time, all of the component's children have been initialized, but the component has not been fully processed. In particular, it has not been sized for layout.

4. Later, to display the application, a `render` event gets triggered, and Flex does the following.
   a. Flex completes all processing required to display the component, including laying out the component.
   b. Flex makes the component visible by setting the `visible` property to `true`.
   c. Flex dispatches the `creationComplete` event on the component. The component has been sized and processed for layout and all properties are set. This event is dispatched only once when the component is created.
   d. Flex dispatches the `updateComplete` event on the component. Flex dispatches additional `updateComplete` events whenever the position, size, or other visual characteristic of the component changes and the component has been updated for display.

You can later remove a component from a container using the `removeChild()` method. The removed child's `parent` property is set to `null`. If you add the removed child to another container, it retains its last known state. If there are no references to the component, it is eventually deleted from memory by the garbage collection mechanism of Adobe Flash Player.

Given this sequence of actions, you should use the events as follows:

- The `preinitialize` event occurs too early in the component life cycle for most initialization activities. It is useful, however, in the rare situations where you must set the properties on a parent before the children are created.

- To configure a component before Flex has determined its visual appearance, use the `initialize` event. For example, use this for setting properties that affect its appearance, height, or width.
- Use the `creationComplete` event for actions that rely on accurate values for the component's size or position when the component is created. If you use this event to perform an action that changes the visual appearance of the component, Flex must recalculate its layout, which adds unnecessary processing overhead to your application.
- Use the `updateComplete` event for actions that must be performed each time a component's characteristics change, not just when the component is created.

Using styles

Flex defines styles for setting some of the characteristics of components, such as fonts, padding, and alignment. These are the same styles as those defined and used with Cascading Style Sheets (CSS). Each visual component inherits many of the styles of its superclasses, and can define its own styles. Some styles in a superclass might not be used in a subclass. To determine the styles that a visual component supports, see the styles section of the page for the component in *Adobe Flex 2 Language Reference*.

You can set all styles in MXML as tag attributes. Therefore, you can set the padding between the border of a Box container and its contents by using the `paddingTop` and `paddingBottom` properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\MXMLStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox id="myVBox1" borderStyle="solid">
        <mx:Button label="Submit"/>
    </mx:VBox>

    <mx:VBox id="myVBox2"
        borderStyle="solid"
        paddingTop="12"
        paddingBottom="12" >

        <mx:Button label="Submit"/>
    </mx:VBox>
</mx:Application>
```

You can also configure styles in ActionScript by using the `setStyle()` method, or in MXML by using the `<mx:Style>` tag. The `setStyle()` method takes two arguments: the style name and the value. The following example is functionally identical to the previous example:

```
<?xml version="1.0"?>
<!-- components\ASStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function initVBox():void {
                myVBox2.setStyle("paddingTop", 12);
                myVBox2.setStyle("paddingBottom", 12);
            }
        ]]>
    </mx:Script>

    <mx:VBox id="myVBox1" borderStyle="solid">
        <mx:Button label="Submit"/>
    </mx:VBox>

    <mx:VBox id="myVBox2"
        borderStyle="solid"
        initialize="initVBox();">

        <mx:Button label="Submit"/>
    </mx:VBox>
</mx:Application>
```

When you use the `<mx:Style>` tag, you set the styles using CSS syntax or a reference to an external file that contains style declarations, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\TagStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        .myStyle {paddingTop: 12; paddingBottom: 12;}
    </mx:Style>

    <mx:VBox id="myVBox1" borderStyle="solid">
        <mx:Button label="Submit"/>
    </mx:VBox>

    <mx:VBox id="myVBox2"
        styleName="myStyle"
        borderStyle="solid">

        <mx:Button label="Submit"/>
    </mx:VBox>
</mx:Application>
```

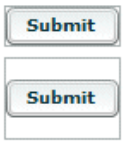
A *class selector* in a style definition, defined as a label preceded by a period, defines a new named style, such as `myClass` in the preceding example. After you define it, you can apply the style to any component by using the `styleName` property. In the preceding example, you apply the style to the second VBox container.

A *type selector* applies a style to all instances of a particular component type.

The following example defines the top and bottom margins for all Box containers:

```
<?xml version="1.0"?>
<!-- components\TypeSelStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        Box {paddingTop: 12; paddingBottom: 12;}
    </mx:Style>

    <mx:VBox id="myVBox" borderStyle="solid">
        <mx:Button label="Submit"/>
    </mx:VBox>

    <mx:Box id="myBox" borderStyle="solid">
        <mx:Button label="Submit"/>
    </mx:Box>
</mx:Application>
```

In Flex, some, but not all, styles are inherited from parent containers to their children and across style types and classes. Because the `borderStyle` style is not inherited by the VBox container, this example yields results that are identical to the previous examples. All of these examples result in the following application:



For more information on styles, see Chapter 18, “Using Styles and Themes,” on page 697.

Using behaviors

A *behavior* is a combination of a trigger paired with an effect. A *trigger* is an action similar to an event, such as a mouse click on a component, a component getting focus, or a component becoming visible. An *effect* is a visible or audible change to the component that occurs over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component. You can define multiple effects for a single trigger.

Flex trigger properties are implemented as CSS styles. In *Adobe Flex 2 Language Reference*, Triggers are listed under the heading "Effects." Flex effects are classes, such as the mx.effects.Fade class.

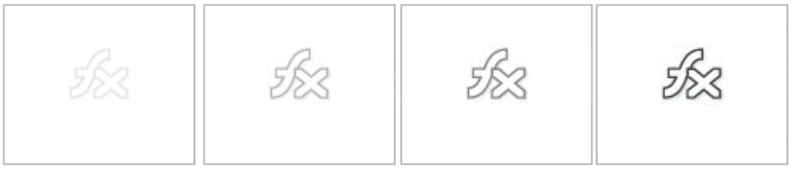
Behaviors let you add animation, motion, and sound to your application in response to some user or programmatic action. For example, you can use behaviors to cause a dialog box to bounce slightly when it receives focus, or to play a sound when the user enters an invalid value.

Because effect triggers are implemented as CSS styles, you can set the trigger properties as tag attributes in MXML, in `<mx:Style>` tags, or in ActionScript by using the `setStyle` function.

To create the behavior, you define a specific effect with a unique ID and bind it to the trigger. For example, the following code creates a fade effect named `myFade` and uses an MXML attribute to bind the effect to the `creationCompleteEffect` trigger of an Image control:

```
<?xml version="1.0"?>
<!-- components\CompIntroBehaviors.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="110"
    height="100"
    backgroundImage="">

    <mx:Fade id="myFade" duration="5000"/>
    <mx:Image
        creationCompleteEffect="{myFade}"
        source="@Embed(source='assets/flexlogo.jpg')"/>
</mx:Application>
```

In this example, the Image control fades in over 5000 milliseconds. The following images show the fade in over the course of time:



For detailed information on using behaviors, see Chapter 17, "Using Behaviors," on page 649.

Applying skins

Skins are graphical style elements that a component uses to control its appearance. Flex components can have of one or more skins. For example, the Button component has eight skins, each for a different button state, such as up, down, disabled, selected, and down, and so on. A control can also have different skins for different subcomponents, for example, the scroll bar has several skins each for the down arrow, up arrow, and thumb.

For more information on skinning, see Chapter 20, "Using Skins," on page 805.

Changing the appearance of a component at run time

You can modify the look, size, or position of a component at run time by using several component properties, styles, or ActionScript methods, including the following:

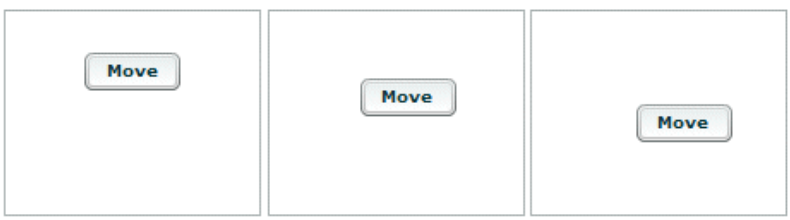
- `x` and `y`
- `width` and `height`
- styles, by using `setStyle(stylename, value)`

You can set the `x` and `y` properties of a component only when the component is in a container that uses absolute positioning; that is, in a Canvas container, or in an Application or Panel container that has the `layout` property set to `absolute`. All other containers perform automatic layout to set the `x` and `y` properties of their children using layout rules.

For example, you could use the `x` and `y` properties to reposition a Button control 15 pixels to the right and 15 pixels down in response to a Button control click, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\ButtonMove.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="150"
    height="120"
    layout="absolute">

    <mx:Script>
        <![CDATA[
            public function moveButton():void {
                myButton.x += 15;
                myButton.y += 15;
            }
        ]]>
    </mx:Script>

    <mx:Button id="myButton"
        x="15"
        y="15"
        label="Move"
        click="moveButton();"/>
</mx:Application>
```

The following example shows the initial image and the results after the user clicks the button each of two times:



In this application, you can move the Button control without concern for other components. However, moving a component in an application that contains multiple components, or modifying one child of a container that contains multiple children, can cause one component to overlap another, or in some other way affect the layout of the application. Therefore, you should be careful when you perform run-time modifications to container layout.

You can set the `width` and `height` properties for a component in any type of container. The following example increases the `width` and `height` of a Button control by 15 pixels each time the user selects it:

```
<?xml version="1.0"?>
<!-- components\ButtonSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="150"
    height="150">

    <mx:Script>
        <![CDATA[
            public function resizeButton():void {
                myButton.height = myButton.height + 15;
                myButton.width = myButton.width + 15;
            }
        ]]>
    </mx:Script>

    <mx:VBox
        borderStyle="solid"
        height="80"
        width="100" >

        <mx:Button id="myButton"
            label="Resize"
            click="resizeButton();"/>
    </mx:VBox>
</mx:Application>
```

This example results in the following progression when the user clicks the button:

If the container that holds the Button does not use absolute positioning, it repositions its children based on the new size of the Button control. The Canvas container and Panel and Application containers with `layout="absolute"` perform no automatic layout, so changing the size of one of their children does not change the position or size of any of the other children.

NOTE

The stored values of `width` and `height` are always in pixels regardless of whether the values were originally set as fixed values, as percentages, or not set at all.

Extending components

Flex 2 SDK provides several ways for you to extend existing components or to create components. By extending a component, you can add new properties or methods to it.

For example, the following MXML component, defined in the file MyComboBox.mxml, extends the standard ComboBox control to initialize it with the postal abbreviations of the states in New England:

```
<?xml version="1.0"?>
<!-- components\myComponents\MyComboBox.mxml -->
<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:dataProvider>
        <mx:String>CT</mx:String>
        <mx:String>MA</mx:String>
        <mx:String>ME</mx:String>
        <mx:String>NH</mx:String>
        <mx:String>RI</mx:String>
        <mx:String>VT</mx:String>
    </mx:dataProvider>
</mx:ComboBox>
```

This example also shows how the MXML compiler lets you use some coding shortcuts. Flex expects the dataProvider to be an array, so you do not have to specify a `<mx:Array>` tag.

After you create it, you can use your new component anywhere in your application by specifying its filename as its MXML tag name, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\MainMyComboBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComps="myComponents.*"
    width="150"
    height="150">

    <MyComps:MyComboBox id="stateNames"/>
</mx:Application>
```

In this example, the new component is in the same directory as your application file, and maps the local namespace, indicated by the asterisk (*), to the MyComps identifier.

Flex lets you create custom components using either of the following methods. The method you choose depends on your application and the requirements of your component.

- Create components as MXML files and use them as custom tags in other MXML files. MXML components provide an easy way to extend an existing component, particularly to modify the behavior of an existing component or add a basic feature to an existing component.

- Create components as ActionScript files by subclassing the UIComponent class or any of its subclasses, and use the resulting classes as custom tags. ActionScript components provide a powerful tool for creating new visual or nonvisual components.

For detailed information on creating custom components, see *Creating and Extending Flex 2 Components*.

CHAPTER 7

Using Data Providers and Collections

7

Several Adobe Flex controls take input from a data provider such as an array or XML object. A Tree control, for example, reads data from a data provider to define the structure of the tree and any associated data assigned to each tree node. Flex controls use collection classes to represent data providers. A collection contains a group of objects, and provides a set of methods that let you access, sort, filter, and modify the items in the collection.

This topic describes data providers and the collection classes. It provides an introduction to using data providers in controls and discusses how to use the collection classes to access and manipulate data. Several topics, including Chapter 12, "Using Data-Driven Controls," on page 439 and Chapter 11, "Using Menu-Based Controls," on page 407 describe Flex controls that use data providers.

Contents

About data providers and collections

Many controls and containers require data for display, for user interaction, or to structure a control such as a menu. To provide this data, you must understand the concepts of data providers and collections and how collections work.

About data providers

Several Flex framework components, including all List based controls, represent data from a *data provider,* an object that contains data required by the control. For example, a Tree control's data provider determines the structure of the tree and any associated data assigned to each tree node, and a ComboBox control's data provider determines the items in the control's drop-down list. Many standard controls, including the ColorPicker and MenuBar controls also get data from a data provider. Controls that display application data are sometimes referred to as *data provider controls*.

The following components use data providers:

- Repeater component
- Chart controls, such as the LineChart control
- ColorPicker control
- ComboBox control
- DataGrid control
- DateField control
- HorizontalList control
- List control
- Menu control
- MenuBar control
- PopUpMenuButton control
- TileList control
- Tree control
- ButtonBar control
- LinkBar control
- TabBar control
- ToggleButtonBar control

The simplest data provider can be an array of strings or objects; for example, you can use the following code to define a static ComboBox control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
        [Bindable]
        public var myArray:Array = ["AL", "AK", "AR"];
    ]]>
  </mx:Script>
  <mx:ComboBox id="myCBO" dataProvider="{myArray}"/>
</mx:Application>
```

However, using a raw data object, such as an Array or Object, as a data provider has limitations:

- Raw objects are often not sufficient if you have data that changes, because the control does not receive a notification of any changes to the base object. The control, therefore, does not get updated until it must be redrawn due to other changes in the application or if the data provider is reassigned; at this time, it gets the data again from the updated Array. In the preceding example, if you programmatically add states to the Array, they do not show up in the ComboBox control unless you reassign the control's `dataProvider` property.
- Raw objects do not provide advanced tools for accessing, sorting, or filtering data. For example, if you use an Array as the data provider, you must use the native Adobe Flash Array methods to manipulate the data.

Flex provides a *collection* mechanism to ensure data synchronization and provide both simpler and more sophisticated data access and manipulation tools. Collections can also provide a consistent interface for accessing and managing data of different types. For more information on collections, see "About collections" on page 167.

For detailed descriptions of the individual controls, see the pages for the controls in *Adobe Flex 2 Language Reference*. For information on programming with many of the data provider-based controls, see Chapter 12, "Using Data-Driven Controls," on page 439.

Data provider types

The Flex framework supports two basic types of data provider:

Linear or list-based data providers are flat data consisting of some number of objects, each of which has the same structure; they are often one-dimensional Arrays, or objects derived from such Arrays, such as ActionScript object graphs. (You can also use an XMLListCollection object as a linear data provider.)

You can use list-based data providers with all data provider controls, and typically use them for all such controls except Tree and most menu-based control instances. If the data provider contents can change dynamically, you typically use the ArrayCollection class and either the IList or ICollectionView interface methods and properties to represent and manipulate these data providers.

Hierarchical data providers consist of cascading levels of often asymmetrical data. Often, the source of the data is an XML object, but the source can be a generic Object or concrete class. You typically use a hierarchical data providers with Flex controls that are designed to display hierarchical data:

- Tree
- Menu, MenuBar, and PopUpMenuButton.

You can also extract data from hierarchical data provider to use in controls such as List or DataGrid that take linear data.

A hierarchical data provider defines a data structure that matches the layout of a tree or cascading menu. For example, a tree often has a root node, with one or more branch or leaf child nodes. Each branch node can hold additional child branch or leaf nodes, but a leaf node is an endpoint of the tree.

The Flex hierarchical data controls use *data descriptor* interfaces to access and manipulate hierarchical data providers, and the Flex framework provides one class, the DefaultDataDescriptor class that implements the required interfaces. If your data provider does not conform to structure required by the default data descriptor, you can create your own data descriptor class that implements the required interfaces.

You can use an ArrayCollection class, XMLListCollection class, ICollectionView interface or IList interface to access and manipulate dynamic hierarchical data.

For more information on using hierarchical data providers, see "Using hierarchical data providers" on page 197.

Data providers and the uid property

Flex data-driven controls, including all controls that are subclasses of List class, use a unique identifier (uid) to track data provider items. Flex can automatically create and manage uids. However, there are circumstances when you must supply your own uid property by implementing the IUID interface, or when supplying your own uid property improves processing efficiency.

NOTE

When Flex creates a UID for an object, such as an item in an ArrayCollection, it adds the UID as an `mx_internal_uid` property of the item. Flex creates `mx_internal_uid` properties for any objects that are dynamic and do not have bindable properties. To avoid having Flex create `mx_internal_uid` properties, the object class should do any of the following things: have at least one property with a `[Bindable]` metadata tag, implement the IUID interface, or have a `uid` property with a value.

If Flex must consider two or more different objects to be identical you must implement the IUID interface so that you can assign the same uid to multiple objects. A typical case where you must implement the IUID interface is an application that uses windowed paged collections. As the cursor moves through the collection, a particular item might be pulled down from the server and released from memory repeatedly. Every time the item is pulled into memory a new object is created to represent the item. If you need to compare items for equality, Flex should consider all objects that represent the same item to be the same "thing."

More common than the case where you must implement the IUID interface is the case where you can improve processing efficiency by doing so. As a general rule, you do not implement the IUID interface if the data provider elements are members of dynamic classes. Flex can automatically create a uid property for these classes. There is still some inefficiency however, so you might consider implementing the IUID interface if processing efficiency is particularly important.

In all other cases, Flex uses the Dictionary mechanism to manage the uid, which might not be as efficient as supplying your own UID.

Because the Object and Array classes are dynamic, you normally do not do anything special for data providers whose items belong to these classes. However, you should consider implementing the IUID if your data provider members belong to custom classes that you define.

The IUID interface contains a single property, `uid`, which is a unique identifier for the class member, and no methods. Flex provides a UIDUtil class that uses a pseudo-random number generator to create an identifier that conforms to the standard GUID format. Although this identifier is not guaranteed to be universally unique, it should be unique among all members of your class. To implement a class that uses the UIDUtil class, such as a Person class that has fields for a first name, last name, and id, you can use the following pattern:

```
package {
  import mx.core.IUID;
  import mx.utils.UIDUtil;

  [Bindable]
  public class Person implements IUID {
    public var id:String;
    public var firstName:String;
    public var lastName:String;
    private var _uid:String;

    public function Person() {
      _uid = createUID();
    }

    public function get uid():String {
      return _uid;
    }

    public function set uid(value:String):void {
      // Do nothing, the constructor created the uid.
    }
  }
}
```

You do not need to use the UIDUtil class in a case where the objects contain a uniquely-identifying field such as an employee ID. In this case, you can use the person's ID as the `uid` property, because the `uid` property values have uniquely identify the object only in the data provider. The following example implements this approach.

```
package
{
  import mx.core.IUID;

  [Bindable]
  public class Person implements IUID {
    public var employee_id:String;
    public var firstName:String;
    public var lastName:String;

    public function get uid(): String {
      return employee_id;
```

```
    }

    public function set uid(value: String): void {
      employee_id=value;
    }
  }
}
```

NOTE	Object cloning does not manage or have a relationship with UIDs, so if you clone something that has an internal UID you must also change that internal UID. UIDs are stored on mx_internal_uid only for dynamic Objects. Instances of data classes that implement IUID will store their UIDs in a .uid property so that is the property that must be changed after cloning. However, dataproviders do not need to implement IUID. If they are instances of data classes that do not implement IUID and are not dynamic objects, the clone technique should work correctly, but most dataprovider items should implement IUID, especially if cloning those objects is not required.

About collections

A *collection* provides a uniform method to access and represent a data provider's data. The collection creates a level of abstraction between Flex components and the data that you use to populate them. The collection interfaces and implementations provide the following features:

- They ensure that a control is properly updated when the underlying data changes. Controls do not update when non-collection data providers change. (They do update to reflect the new data the next time they are refreshed.) If the data provider is a collection, the controls update immediately after the collection change occurs.
- They provide mechanisms for handling paged data coming from remote data providers that may not initially be available and may arrive over a period of time.
- They provide a consistent set of operations on the data, independent of those provided by the raw data provider objects. For example, you could create a custom class that implements the IList interface and lets you insert and delete objects by using an index into the collection, independent of whether the underlying data is, for example, in an array or an object.
- Collections that conform to the ICollectionView interface provide a specific *view* of the data that can be in sorted order, or filtered by a developer-supplied method. These limitations affect only the collection's view, and do not change the underlying data.
- You can use a single collection to populate multiple components from the same data provider.

- You can use collections to switch data providers for a component at run time, and to modify a data provider so that changes are reflected by all components that use the data provider.
- You can use collection methods to access data in the underlying data object.

NOTE

If you use a raw object, such as an Array, as a control's data provider, Flex automatically wraps the object in a collection wrapper. The control does not automatically detect changes that are made directly to the raw object. A change in the length of an array, for example, does not result in an update of the control. You can, however, use an object proxy, a listener interface, or the itemUpdated property to notify the view of certain changes.

Collection interfaces

The Flex framework collection model uses these interfaces to define how a collection represents data and provides access to it:

Interface	Description
IList	A direct representation of items organized in an ordinal fashion. The interface presents the data from the data provider in the same order as it exists in the provider, and provides access and manipulation methods based on an index. The IList class does not provide sorting, filtering, or cursor functionality.
ICollectionView	A view of a collection of items. You can modify the view to show the data in sorted order and to show a subset of the items in the data provider, as specified by a filter function. A class that implements this interface can use an IList interface as the underlying collection. The interface provides access to an IViewCursor object for access to the items.
IViewCursor	Enumerates an ICollectionView object bidirectionally. The *view cursor* provides find, seek, and bookmarking capabilities, and lets you modify the underlying data (and the view) by inserting and removing items.

The IList and ICollectionView interfaces provide two alternate methods for accessing and changing data. The IList interface is simpler; it provides add, set, get, and remove operations that operate directly on linear data.

The ICollectionView interface (also called the collection view) provides a more complex set of operations than the IList interface, and is appropriate when the underlying data may not be organized linearly. Its data access techniques, however, are more complex than those of the IList interface, so you should use the IList interface if you need only simple, indexed access to a linear collection. The collection view can represent a subset of the underlying data, as determined by a sort or filter operation.

The standard Flex collection classes, ArrayCollection and XMLListCollection, implement both the collection view (ICollectionView) interface and the IList interface.

For information on using the IList interface methods and property, see "Using IList interface methods and properties" on page 174. For information on using the ICollectionView interface methods and properties, Including the IViewCursor object that the interface exposes, see "Using ICollectionView interface methods and properties" on page 176.

Collection classes

The following table describes the public classes in the mx.collections package. It does not include constant classes, event and error classes. For complete reference information on collection-related classes, see the collections and collections.errors packages, and the CollectionEvent and CollectionEventKind classes in *Adobe Flex 2 Language Reference*.

Class	Description
ArrayCollection	Implements the IList and ICollectionView interfaces for use with array-based data providers.
XMLListCollection	Implements the IList and ICollectionView interfaces, and a subset of XMLList methods for use with XML-based data providers.
CursorBookmark	Represents the position of an IViewCursor instance within an ICollectionView instance. You can save a view cursor position in a CursorBookmark object and use the object to return the view cursor to the position at a later time.
Sort	Provides the information and methods required to sort an ICollectionView instance.
SortField	Provides properties and methods that determine how a specific field affects how data is sorted in an ICollectionView instance.
ItemResponder	(Used only if the data source is remote.) Handles cases when requested data is not yet available.
ListCollectionView	Adapts an object that implements the IList interface to the ICollectionView interface so that it can be passed to anything that expects an IList or an ICollectionView. This class is the superclass of the ArrayCollection and XMLListCollection classes

Specifying data providers in MXML applications

The Flex framework lets you specify and access data providers in many ways. For example, as with any MXML control, you can define the data provider by using an `<mx:dataProvider>` property child tag of a control, or you can define the data provider in an ActionScript assignment. All access techniques belong to one of three major patterns:

- Using a raw data object, such as an Array.
- Using a collection class object directly. This pattern is particularly useful for local collections where object reusability is less important.
- Using an interface. This pattern provides the maximum of independence from the underlying implementation.

Using a raw data object as a data provider

You can use a raw data object, such as an Array object, directly as a data provider any time the data is static. For example, you could use an array for a static list of U.S. Postal Office state designators. Do not use the data object directly as the dataProvider property of a control if the object contents can change dynamically, for example in response to user input or programmatic processing.

The result returned by an HTTPService or WebService often is an Array, and if you treat that data as read-only, you can use the Array directly as a data provider. (RemoteObject classes often return ArrayCollections.)

List-based controls, however, internally turn Array-based data providers into collections, so there is no performance advantage of using an Array as the data provider. If you pass an Array to multiple controls, it is more efficient to convert the Array into a collection when the data is received, and then pass the collection to the controls.

To use a raw data object, assign the object to the control's `dataProvider` control, or define the object directly in the body of the control MXML tag, as shown in the example in "About data providers" on page 162.

Using a collection object directly

You can use an object that implements the ICollectionView or IList interface as a data provider directly in an MXML control by assigning it to the component's `dataProvider` property. This technique is more direct than using an interface and is appropriate if the data provider always belongs to a specific collection class.

For list-based controls, you often use an ArrayCollection object as the data provider, and populate the ArrayCollection using an Array. The following example shows a data provider that specifies the data for a ComboBox control:

```
<?xml version="1.0"?>
<!-- dpcontrols\ArrayCollectionInComboBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:ComboBox id="myCB">
        <mx:ArrayCollection id="stateArray">
            <mx:Object label="AL" data="Montgomery"/>
            <mx:Object label="AK" data="Juneau"/>
            <mx:Object label="AR" data="Little Rock"/>
        </mx:ArrayCollection>
    </mx:ComboBox>
</mx:Application>
```

In this example, the `dataProvider` default property of the ComboBox defines an ArrayCollection object whose `source` default property is an array of Objects, each of which has a label and data field. (The ArrayCollection source property's takes an Array object, so there is no need to use an `<mx:Array>` tag in this example.)

The following example uses ActionScript to declare and create the ArrayCollection object:

```
<?xml version="1.0"?>
<!-- dpcontrols\ArrayCollectionInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initData()">
    <mx:Script>
        <![CDATA[
            import mx.collections.*;
            [Bindable]
            public var stateArray:ArrayCollection;

            public function initData():void {
                stateArray=new ArrayCollection(
                [{label:"AL", data:"Montgomery"},
                {label:"AK", data:"Juneau"},
                {label:"AR", data:"Little Rock"}]);
            }
        ]]>
    </mx:Script>

    <mx:ComboBox id="myComboBox" dataProvider="{stateArray}"/>
</mx:Application>
```

After you define the ComboBox control, the `dataProvider` property of the ComboBox control provides access to the collection that represents the underlying data provider source object, and you can use the property to modify the data provider. If you add the following button to the preceding code, for example, you can click the button to add the label and data for Arizona to the end of the list in the ComboBox.

```
<mx:Button label="Add AZ"
click="myComboBox.dataProvider.addItem({'label':'AZ',
'data':'Phoenix'});"/>
```

It is generally a better practice, and highly recommended, to reference the collection directly, as in the following example:

```
<mx:Button label="Add AZ"
  click="stateArray.addItem({'label':'AZ', 'data':'Phoenix'});"/>
```

Accessing the data provider using collection interfaces

If you know that a control's data provider can always be represented by a specific collection class, use the class directly, as shown in "Using a collection object directly". If your code might be used with different underlying collection types, then you should use the ICollectionView Interface in your application code, as shown in the following meta-example:

```
public var myICV:ICollectionView = indeterminateCollection;
<mx:ComboBox id="cb1" dataProvider="{myICV}" initialize="sortICV()"/>
```

You can then manipulate the interface as needed to select data for viewing, or to get and modify the data in the underlying data source. For more detailed information the techniques provided by the collection interfaces, see Using IList interface methods and properties on page 174 and Using ICollectionView interface methods and properties on page 176.

Example: Using a simple data provider

The following sample code shows how you can use basic collection classes to represent and manipulate an Array for use in a control. This example shows the following features:

- Using an ArrayCollection to represent data in an array
- Sorting the ArrayCollection
- Inserting data in the ArrayCollection.

This example also shows the insertion's effect on the Array and the ArrayCollection representation of the Array:

```
<?xml version="1.0"?>
<!-- dpcontrols\SimpleDP.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
initialize="sortAC()">
    <mx:Script>
        <![CDATA[
            import mx.collections.*;

            // Function to sort the ArrayCollection in descending order.
            public function sortAC():void {
                var sortA:Sort = new Sort();
                sortA.fields=[new SortField("label")];
                myAC.sort=sortA;
                //Refresh the collection view to show the sort.
                myAC.refresh();
            }
            // Function to add an item in the ArrayCollection.
           // Data added to the view is also added to the underlying Array.
            // The ArrayCollection must be sorted for this to work.
            public function addItemToMyAC():void {
                myAC.addItem({label:"MD", data:"Annapolis"});
            }
        ]]>
    </mx:Script>

    <!-- An ArrayCollection with an array of objects -->
    <mx:ArrayCollection id="myAC">
        <!-- Use an mx:Array tag to associate an id with the array. -->
        <mx:Array id="myArray">
            <mx:Object label="MI" data="Lansing"/>
            <mx:Object label="MO" data="Jefferson City"/>
            <mx:Object label="MA" data="Boston"/>
            <mx:Object label="MT" data="Helena"/>
            <mx:Object label="ME" data="Augusta"/>
            <mx:Object label="MS" data="Jackson"/>
            <mx:Object label="MN" data="Saint Paul"/>
        </mx:Array>
    </mx:ArrayCollection>

    <mx:HBox width="100%">
        <!-- A ComboBox populated by the underlying Array object.
            This control shows that Array retains its original order
            and MD is inserted at the end of the Array. -->
        <mx:ComboBox id="cb2" rowCount="10" dataProvider="{myArray}"/>
        <!-- A ComboBox populated by the collection view of the Array. -->
        <mx:ComboBox id="cb1" rowCount="10" dataProvider="{myAC}"/>
        <mx:Button id="b1" label="Add MD" click="addItemToMyAC();"/>
```

```
    </mx:HBox>
</mx:Application>
```

Using IList interface methods and properties

The IList interface provides simple methods for indexed access to linear data. The interface provides a direct representation of the underlying data provider. Thus, any operation that changes the collection changes the data provider in a similar manner: if you insert a item as the third item in the collection, it also is the third item in the underlying data source.

NOTE	If you use the ICollectionView interface to sort or filter a collection, do not use the IList interface to manipulate the data, as the results are indeterminate.

The interface includes properties and methods that let you do the following:

- Get, set, add, or remove an item at a specific index into the collection.
- Add an item at the end of the collection.
- Get the index of a specific item in the collection.
- Remove all items in the collection.
- Get the length of the collection.

You can use the IList interface methods and properties directly on any of the following classes or properties:

- ArrayCollection class
- XMLList class
- `dataProvider` property of standard Flex controls.

The following sample code uses an ArrayCollection object and its implementation of the IList interface methods to display an array of elements in a ComboBox control. For an example that uses IList interface methods to manage an ArrayCollection of objects with multiple fields, see "Example: Modifying data in DataGrid control" on page 194.

In the following example the Array data source initially consists of the following elements:

```
"AZ", "MA", "MZ", "MN", "MO", "MS"
```

When you click the Button control, the application uses the `length` property and several methods of the IList interface to do the following:

- Change the data in the array and the displayed data in the ComboBox control to a correct alphabetical list of the U.S. ZIP codes for states that start with M:

  `MA, ME, MI, MN, MO, MS, MT`

- Display in a TextArea control information about the tasks it performed and the resulting array.

The code includes comments that describe the changes to the data provider.

```
<?xml version="1.0"?>
<!-- dpcontrols\UseIList.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initData()">

    <mx:Script>
        <![CDATA[
            import mx.collections.*;

            // The data provider is an Array of Strings
            public var myArray:Array = ["AZ", "MA", "MZ", "MN", "MO", "MS"];
            // Declare an ArrayCollection that represents the Array.
            [Bindable]
            public var myAC:ArrayCollection;

            //Initialize the ArrayCollection.
            public function initData():void {
                myAC = new ArrayCollection(myArray);
            }

            // The function to change the collection, and therefore
            // the Array.
            public function changeCollection():void {
                // Get the original collection length.
                var oldLength:int=myAC.length;

                // Remove the invalid first item, AZ.
                var removedItem:String=String(myAC.removeItemAt(0));
               // Add ME as the second item. (ILists used 0-based indexing.)
                myAC.addItemAt("ME", 1);
                // Add MT at the end of the Array and collection.
                myAC.addItem("MT");
                // Change the third item from MZ to MI.
                myAC.setItemAt("MI", 2);
                // Get the updated collection length.
                var newLength:int=myAC.length;
                // Get the index of the item with the value ME.
                var addedItemIndex:int=myAC.getItemIndex("ME");
```

```
                // Get the fifth item in the collection.
                var index4Item:String=String(myAC.getItemAt(4));

                // Display the information in the TextArea control.
                ta1.text="Start Length: " + oldLength + ". New Length: " +
                    newLength;
                ta1.text+=".\nRemoved " + removedItem;
                ta1.text+=".\nAdded ME at index " + addedItemIndex;
                ta1.text+=".\nThe item at index 4 is " + index4Item + ".";
                // Show that the base Array has been changed.
                ta1.text+="\nThe base Array is: " + myArray.join();
            }
        ]]>
    </mx:Script>

    <mx:ComboBox id="myCB" rowCount="7" dataProvider="{myAC}"/>
    <mx:TextArea id="ta1" height="75" width="300"/>
    <mx:Button label="rearrange list" click="changeCollection();"/>
</mx:Application>
```

Using ICollectionView interface methods and properties

The ICollectionView interface represents a *view* of the underlying data source as a collection of items. It has the following major features:

- You can modify the view to show the data in sorted order or to show a subset of the items in the data provider without changing the underlying data. For more information, see "Sorting and filtering data for viewing" on page 177
- You can access the collection data using a *cursor*, which lets you move through the collection, use bookmarks to save specific locations in the collection, and insert and delete items in the collection (and therefore in the underlying data source). For more information, see "Using the IViewCursor interface" on page 180
- You can use ICollectionView objects to represent remote data that might not initially be available, or where parts of the data set might become available at different times. For more information, see "Using collection change notifications" on page 191 and "Using remote data providers" on page 213

You can use the ICollectionView interface methods and properties directly on any of the following classes or properties:

- ArrayCollection class
- XMLListCollection class

- `dataProvider` property of all standard Flex controls *except* those that are subclasses of the NavBar class (ButtonBar, LinkBar, TabBar, ToggleButtonBar)

The following sections describe the basic ICollectionView operations using a list-based collection, but can also apply to similar operations on a hierarchical collection.

Sorting and filtering data for viewing

The ICollectionView interface lets you sort and filter the data in the data provider so that the view represented by the collection is a reordered subset of the underlying data. These operations have no effect on the data provider contents, only on the subset that the collection view represents, and therefore what any control that uses the collection displays.

Sorting

The Sort class lets you sort data in the collection. You can specify multiple fields to use in sorting the data, require that the resulting entries be unique, and specify a custom comparison function to use for ordering the sorted output. You can also use the Sort class to find items in a collection. When you create a Sort class, or change its properties, you must call the `refresh()` method on the collection to show the results.

You use the SortField class to specify the fields to use in the sort. You create the SortField objects and put them in the Sort class object's `fields` array.

The following shows a function to sort a collection. In this example, myAC is a collection view of an Array of objects containing label and name fields. The primary sort is a descending, case-insensitive sort on the area field and the secondary sort is an ascending case-sensitive sort on the label field. You might call it as the initialize event handler for a control that must display a specific sorted view of a collection.

```
//Sort the ICollectionView in descending order.
public function sortAC():void {
  //Create a Sort object.
  var sortA:Sort = new Sort();
  // Sort first on the area field, then the label field.
  // The second parameter specifies that the sort is case-insensitive.
  // A true third parameter specifies a descending sort.
  sortA.fields=[new SortField("area", true, true),
              new SortField("label")];
  myAC.sort=sortA;
  //Refresh the collection view to show the sort.
  myAC.refresh();
}
```

Filtering

You use a filter function limit the ICollection view to a subset of the data provider source. The function must take a single Object parameter, which corresponds to a collection item, and must return a Boolean value specifying whether to include the item in the collection view. As with sorting, when you specify or change the filter function, you must call the `refresh()` method on the collection to show the filtered results. To limit a collection view of an array of strings to contain only strings starting with M, for example, use the following filter function:

```
public function stateFilterFunc(item:Object):Boolean
{
  return item >= "M" && item < "N";
}
```

Example: Sorting and filtering an ArrayCollection

The following example shows the use of the filter function and a sort together. You can use the buttons to sort, or filter the collection, or you can do both. Use the Reset button to restore the collection view to its original state.

```
<?xml version="1.0"?>
<!-- dpcontrols\SortFilterArrayCollection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600">
    <mx:Script>
        <![CDATA[
            import mx.collections.*;

            // Function to sort the ICollectionView
            // in ascending order.
            public function sortAC():void {
                var sortA:Sort = new Sort();
                sortA.fields=[new SortField("label")];
                myAC.sort=sortA;
                //Refresh the collection view to show the sort.
                myAC.refresh();
            }

            // Function to filter out all items with labels
            // that are not in the range of M-N.
            public function stateFilterFunc(item:Object):Boolean {
                return item.label >= "M" && item.label < "O";
            }

            // Function to apply the filter function the ICollectionView.
            public function filterAC():void {
                myAC.filterFunction=stateFilterFunc;
                //Refresh the collection view to apply the filter.
                myAC.refresh();
            }
```

```
            // Function to Reset the view to its original state.
            public function resetAC():void {
                myAC.filterFunction=null;
                myAC.sort=null;
                //Refresh the collection view.
                myAC.refresh();
            }

        ]]>
    </mx:Script>

    <!-- An ArrayCollection with an array of objects. -->
    <mx:ArrayCollection id="myAC">
        <mx:Array id="myArray">
            <mx:Object label="LA" data="Baton Rouge"/>
            <mx:Object label="NH" data="Concord"/>
            <mx:Object label="TX" data="Austin"/>
            <mx:Object label="MA" data="Boston"/>
            <mx:Object label="AZ" data="Phoenix"/>
            <mx:Object label="OR" data="Salem"/>
            <mx:Object label="FL" data="Tallahassee"/>
            <mx:Object label="MN" data="Saint Paul"/>
            <mx:Object label="NY" data="Albany"/>
        </mx:Array>
    </mx:ArrayCollection>

    <!-- Buttons to filter, sort, or reset the view in the second ComboBox
            control. -->
    <mx:HBox width="100%">
        <mx:Button id="sortButton" label="Sort" click="sortAC();"/>
        <mx:Button id="filterButton" label="Filter" click="filterAC();"/>
        <mx:Button id="resetButton" label="Reset" click="resetAC();"/>
    </mx:HBox>
    <mx:HBox width="100%">
        <!-- A ComboBox populated by the underlying Array object.
            This control shows that Array retains its original order. -->
        <mx:ComboBox id="cb2" rowCount="10" dataProvider="{myArray}"/>
        <!-- A ComboBox populated by the collection view of the Array. -->
        <mx:ComboBox id="cb1" rowCount="10" dataProvider="{myAC}"/>
    </mx:HBox>
</mx:Application>
```

For a more complex example of sorting a DataGrid control, which both does an initial sort of the data and does a custom sort when you click a column heading, see "Example: Sorting a DataGrid on multiple columns" on page 476.

Using the IViewCursor interface

The ICollectionView interface includes a `createCursor()` method that returns an IViewCursor object, also called a cursor, that you can use to traverse the items in the view and to access and modify data in the collection. A *cursor* is a position indicator; it points to a particular item in the collection. You can use IViewCursor methods and properties to perform the following operations:

- Move the cursor backward or forward.
- Move the cursor to specific items.
- Get the item at a cursor location.
- Add, remove, and change items.
- Save a cursor position by using a bookmark, and return to it later.

When you use standard Flex collection classes, ArrayCollection and XMLListCollection, you use the IViewCursor interface directly, you do not reference an object instance, as shown in the following code snippet:

```
public var myAC:ICollectionView = new ArrayCollection(myArray);
public var myCursor:IViewCursor;
.
.
myCursor=myAC.createCursor();
```

Manipulating the view cursor

The IViewCursor interface includes the following methods and properties for moving the cursor:

- The `moveNext()` and `movePrevious()` methods move the cursor forward and backward by one item. Use the `beforeFirst` and `afterLast` properties to check whether you've reached the bounds of the view. The following example moves the cursor to the last item in the view:

  ```
  while (! myCursor.afterLast) {
       myCursor.moveNext();
  }
  ```

- The `findAny()`, `findFirst()`, and `findLast()`, methods move the cursor to an item that matches the parameter. Before you can use these methods, you must apply a Sort to the ICollectionView implementation (because the functions use Sort methods).

  If it is not important to find the first occurrence of an item or last occurrence of an item in a non-unique index, the `findAny()` method can be somewhat more efficient than either the `findFirst()` or the `findLast()` method.

If the associated collection is remote, and not all of the items are cached locally, the find methods begin an asynchronous fetch from the remote collection; if a fetch is already in progress, they wait for it to complete before making another fetch request.

The following example finds an item inside a collection of simple objects, in this case, an ArrayCollection of state ZIP Code strings. It creates a default Sort object, applies it to an ArrayCollection object, and finds the first instance of the string `"MZ"` in a simple array of strings:

```
var sortD:Sort = new Sort();
// The null first parameter on the SortField constructor specifies a
// collection of simple objects (String, numeric, or Boolean values).
// The true second parameter specifies a case-insensitive sort.
sortD.fields = [new SortField(null, true)];
myAC.sort=sortD;
myAC.refresh();
myCursor.findFirst("MZ");
```

To find a complex object, the `findFirst()` method can search on multiple sort fields. You cannot, however, skip fields in the parameter of any of the find methods. If an object has three fields, for example, you can specify any of the following field combinations in the parameter: 1, 1,2, 1,2,3, but you cannot specify only fields 1 and 3.

Both of the following lines will find an object with the label value `"ME"` and data value `"Augusta"`:

```
myCursor.findFirst({label:"ME"});
myCursor.findFirst({label:"ME", data:"Augusta"});
```

- The `seek()` method moves the cursor to a position relative to a bookmark. You use this method to move the cursor to the first or last item in a view, or to move to a bookmark position that you have saved. For more information on using bookmarks and the `seek()` method, see "Using bookmarks" on page 183

Getting, adding and removing items

The IViewCursor interface includes the following methods and properties for accessing and changing data in the view:

- The `current` property is a reference to the item at the current cursor location.
- The `insert()` method inserts an item before the current cursor location. Note, however, that if the collection is sorted, (for example, to do a `find()`) the sort moves the item to the sorted order location, not the cursor location.
- The `remove()` method removes the item at the current cursor location; if the removed item is not the last item, the cursor then points to the location after the removed item.

The following example shows the results of using the `insert()` and `remove()` on the `current` property:

```
<?xml version="1.0"?>
<!-- dpcontrols\GetAddRemoveItems.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        initialize="initData();">
    <mx:Script>
        <![CDATA[
            import mx.collections.*;
           public var myArray:Array = [{label:"MA", data:"Massachusetts"},
            {label:"MN", data:"Minnesota"}, {label:"MO", data:"Missouri"}];
            [Bindable]
            public var myAC:ArrayCollection;
            public var myCursor:IViewCursor;

            // Initialize the ArrayCollection when you
            // initialize the application.
            public function initData():void {
                myAC = new ArrayCollection(myArray);
            }

            // The function to change the collection,
            // and therefore the Array.
            public function testCollection():void {
                // Get an IViewCursor object for accessing the collection
data.
                myCursor=myAC.createCursor();
               ta1.text="At start. cursor is at: " + myCursor.current.label;
                var removedItem:String=String(myCursor.remove());
               ta1.text+="\nAfter removing the current item, the cursor is
at: "
                    + myCursor.current.label;
                myCursor.insert({label:"ME", data:"Augusta"});
                ta1.text+="\nAfter adding an item, the cursor is at: "
                    + myCursor.current.label;
            }
        ]]>
    </mx:Script>

    <mx:ComboBox id="myCB" rowCount="7" dataProvider="{myAC}"/>
    <mx:TextArea id="ta1" height="75" width="350"/>
    <mx:Button label="run test" click="testCollection();"/>
</mx:Application>
```

Using bookmarks

You use a bookmark to save a cursor location for later use. You can also use the built-in `FIRST` and `LAST` bookmark properties to move the cursor to the first or last item in the view.

To create and use a bookmark:

1. Move the cursor to a desired location in the view.
2. Assign the current value of the `bookmark` property to a variable, as in the following line:

   ```
   var myBookmark:CursorBookmark=myIViewCursor.bookmark;
   ```
3. Do some operations that might move the cursor.
4. When you must return to the bookmarked cursor location (or to a specific offset from the bookmarked location), call the IViewCursor `seek()` method, as in the following line:

   ```
   myIViewCursor.seek(myBookmark);
   ```

The following example counts the number of items in a collection between the selected item in a ComboBox and the end of the collection, and then returns the cursor to the initial location:

```
<?xml version="1.0"?>
<!-- dpcontrols\UseBookmarks.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        initialize="run();">
    <mx:Script>
        <![CDATA[
            import mx.collections.*;
            private var myCursor:IViewCursor;

            // Initialize variables.
            public function run():void {
                // Initialize the cursor.
                myCursor=myAC.createCursor();
                // The findFirst() method, used in
                // countFromSelection() requires a
                // sorted view.
                var sort:Sort = new Sort();
                sort.fields=[new SortField("label")];
                myAC.sort=sort;
                //You must refresh the view to apply the sort.
                myAC.refresh();
            }

            // Count the items following the current
            // cursor location.
            public function countLast(theCursor:IViewCursor):int {
                var counter:int=0;
                // Set a bookmark at the current cursor location.
                var mark:CursorBookmark=theCursor.bookmark;
                // Move the cursor to the end of the Array.
                // The moveNext() method returns false when the cursor
                // is after the last item.
                    while (theCursor.moveNext()) {
                    counter++;
                }
                // Return the cursor to the initial location.
                theCursor.seek(mark);
                return counter;
            }

            // Function triggered by ComboBox change event.
            // Calls the countLast() function to count the
            // number of items to the end of the collection.
            public function countFromSelection():void {
                myCursor.findFirst(myCB.selectedItem);
                var count:int = countLast(myCursor);
```

```
                ta1.text += myCursor.current.label + " is " + count +
                            " from the last item.\n";
            }
        ]]>
    </mx:Script>

    <!-- The data provider, an ArrayCollection with an array of objects. -->
    <mx:ArrayCollection id="myAC">
        <mx:Object label="MA" data="Boston"/>
        <mx:Object label="ME" data="Augusta"/>
        <mx:Object label="MI" data="Lansing"/>
        <mx:Object label="MN" data="Saint Paul"/>
        <mx:Object label="MO" data="Jefferson City"/>
        <mx:Object label="MS" data="Jackson"/>
        <mx:Object label="MT" data="Helena"/>
    </mx:ArrayCollection>

    <mx:ComboBox id="myCB" rowCount="7" dataProvider="{myAC}"
change="countFromSelection();"/>
    <mx:TextArea id="ta1" height="200" width="175"/>
</mx:Application>
```

Example: Updating an Array Using ICollectionView interface methods and properties

The following example uses the ICollectionView methods and properties of an ArrayCollection object to display an array with the following elements in a ComboBox control:

```
"AZ", "MA", "MZ", "MN", "MO", "MS"
```

When you click the Update View button, the application uses the `length` property and several methods of the ICollectionView interface to do the following:

- Changes the data in the array and the displayed data in the ComboBox control to a correct alphabetical list of the U.S. ZIP codes for states that start with M:

  ```
  MA, ME, MI, MN, MO, MS, MT
  ```

- Saves a bookmark that points to the ME item that it adds, and later restores the cursor to this position.
- Displays in a TextArea control information about the tasks it performed and the resulting array.

When you click the Sort button, the application reverses the order of the items in the view, and limits the viewed range to ME–MO.

When you click the Reset button, the application resets the data provider array and the collection view.

```
<?xml version="1.0"?>
<!-- dpcontrols\UpdateArrayViaICollectionView.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        initialize="initData();">

    <mx:Script>
        <![CDATA[
            import mx.collections.*;
            // The data provider is an array of Strings.
            public var myArray:Array = ["AZ", "MA", "MZ", "MN", "MO", "MS"];
            // Declare an ArrayCollection that represents the Array.
            // The variable must be bindable so the ComboBox can update
properly.
            [Bindable]
            public var myAC:ArrayCollection;
            //Boolean flag to ensure the update routine hasn't been run
before.
            public var runBefore:Boolean=false;

            //Initialize the ArrayCollection the application initializes.
            public function initData():void {
                myAC = new ArrayCollection(myArray);
            }

            // The function to change the collection.
            public function changeCollection():void {
                //Running this twice without resetting causes an error.
                if (! runBefore) {
                    runBefore=true;
                  // Get an IViewCursor object for accessing the collection
data.
                    var myCursor:IViewCursor=myAC.createCursor();
                    // Get the original collection length.
                    var oldLength:int=myAC.length;

                   // The cursor is initially at the first item; delete it.
                    var removedItem:String=String(myCursor.remove());

                    // Add ME as the second item.
                    // The cursor is at the (new) first item;
                    // move it to the second item.
                    myCursor.moveNext();
                    // Insert ME before the second item.
                    myCursor.insert("ME");

                    // Add MT at the end of the collection.
                   //Use the LAST bookmark property to go to the end of the
```

```
view.
                    // Add an offset of 1 to position the cursor after the
last item.
                    myCursor.seek(CursorBookmark.LAST, 1);
                    myCursor.insert("MT");
                    // Change MZ to MI.
                    // The findFirst() method requires a sorted view.
                    var sort:Sort = new Sort();
                    myAC.sort=sort;
                    // Refresh the collection view to apply the sort.
                    myAC.refresh();
                   // Make sure there is a MZ item, and no MI in the array.
                     if (myCursor.findFirst("MZ") &&
!myCursor.findFirst("MI")) {
                       // The IViewCursor does not have a replace operation.
                        // First, remove "MZ".
                        myCursor.remove();
                        // Because the view is now sorted, the insert puts
this item
                       // in the right place in the sorted view, but at the
end of
                        // the underlying Array data provider.
                        myCursor.insert("MI");
                    }

                    // Get the updated collection length.
                    var newLength:int=myAC.length;

                    // Set a bookmark at the item with the value ME,
                    myCursor.findFirst("ME");
                    var MEMark:CursorBookmark=myCursor.bookmark;
                    // Move the cursor to the last item in the Array.
                    myCursor.seek(CursorBookmark.LAST);
                    // Get the last item in the collection.
                    var lastItem:String=String(myCursor.current);
                    // Return the cursor to the bookmark position.
                    myCursor.seek(MEMark);
                    // Get the item at the cursor location.
                    var MEItem:String=String(myCursor.current);

                    // Display the information in the TextArea control.
                   ta1.text="Start Length: " + oldLength + ". End Length: "
                                    + newLength;
                    ta1.text+=".\nRemoved " + removedItem;
                    ta1.text+=".\nLast Item is " + lastItem;
                    ta1.text+=".\nItem at MEMark is " + MEItem;
                    // Show that the base Array has been changed.
                    // Notice that the Array is NOT in sorted order.
                    ta1.text+="\nThe base Array is: " + myArray.join();
                } // End runBefore condition
```

```
            }

           // Filter function used in the sortICV method to limit the range.
            public function MEMOFilter(item:Object):Boolean {
                return item >= "ME" && item <= "MO";
            }

            // Sort the collection view in descending order,
            // and limit the items to the range ME - MO.
            public function sortICV():void {
                var sort:Sort = new Sort();
                sort.fields=[new SortField(null, false, true)];
                myAC.filterFunction=MEMOFilter;
                myAC.sort=sort;
                // Refresh the ArrayCollection to apply the sort and filter
                // function.
                myAC.refresh();
                //Call the ComboBox selectedIndex() method to replace the
"MA"
                //in the display with the first item in the sorted view.
                myCB.selectedIndex=0;
                ta1.text="Sorted";
            }

            //Reset the Array and update the display to run the example
again.
            public function resetView():void {
                myArray = ["AZ", "MA", "MZ", "MN", "MO", "MS"];
                myAC = new ArrayCollection(myArray);
                ta1.text="Reset";
                runBefore=false;
            }
        ]]>
    </mx:Script>

    <mx:ComboBox id="myCB" rowCount="7" dataProvider="{myAC}"/>
    <mx:TextArea id="ta1" height="75" width="300"/>
    <mx:HBox>
        <mx:Button label="Update View" click="changeCollection();"/>
        <mx:Button label="Sort View" click="sortICV();"/>
        <mx:Button label="Reset View" click="resetView();"/>
    </mx:HBox>
</mx:Application>
```

Using events and update notifications

The Flex collection mechanism includes events that represent changes to the collection and provides methods to control the delivery of events. The following sections describe these events, and discuss the ways you can control event delivery.

Using collection events

Flex includes several classes that you use in collection-related events:

- Classes that implement the IList or ICollectionView interface dispatch a CollectionEvent (mx.events.CollectionEvent) class event whenever the collection changes. All collection events have the `type` property value `CollectionEvent.COLLECTION_CHANGE`.
- The CollectionEvent object includes a `kind` property that indicates the way in which the collection changed; you can determine the change by comparing the `kind` property value with the CollectionEventKind constants, for example, `UPDATE`.
- The CollectionEvent object includes an `items` property that is an Array of objects whose type varies depending on the event kind. For ADD and REMOVE kind events, the array contains the added or removed items. For UPDATE events, the items property contains an Array of PropertyChangeEvent event objects. This object's properties indicate the type of change and the property value before and after the change.
- The `PropertyChangeEvent` class `kind` property indicates the way in which the property changed; you can determine the change type by comparing the `kind` property value with the PropertyChangeEventKind constants, for example, `UPDATE`.
- Classes that implement the IViewCursor interface dispatch a FlexEvent class event with a type of the `type` property value of `mx.events.FlexEvent.CURSOR_UPDATE` when the cursor position changes.

You use collection events to monitor changes to a collection to update the display. For example, if a custom control uses a collection as its data provider, and you want the control to update dynamically and display the revised data each time the collections changes, the control can monitor the collection events and update accordingly.

You could, for example, build a simple rental-car reservation system that uses collection events. This application uses `COLLECTION_CHANGE` event type listeners for changes to its `reservations` and `cars` data collections.

The CollectionEvent listener method, named `reservationsChanged`, tests the event `kind` field and does the following:

- If the event `kind` property is `ADD`, iterates through the objects in the event's `items` property and calls a function to update the reservation information display with boxes that display the time span of each reservation.
- If the event `kind` property is `REMOVE`, iterates through the objects in the event's `items` property and calls a function to remove the reservation box for each item.
- If the event `kind` property is `UPDATE`, iterates through the `PropertyChangeEvent` objects in the event's `items` property and calls the update function to update each item.
- If the event `kind` property is `RESET`, calls a function to reset the reservation information.

The following examples shows the `reservationsChanged` CollectionEvent event listener function:

```
private function reservationsChanged(event:CollectionEvent):void {
  switch (event.kind) {
    case CollectionEventKind.ADD:
      for (var i:uint = 0; i < event.items.length; i++) {
          updateReservationBox(Reservation(event.items[i]));
      }
      break;

    case CollectionEventKind.REMOVE:
      for (var i:uint = 0; i < event.items.length; i++) {
        removeReservationBox(Reservation(event.items[i]));
      }
      break;

    case CollectionEventKind.UPDATE:
      for (var i:uint = 0; i < event.items.length; i++) {
        if (event.items[i] is PropertyChangeEvent) {
          if (PropertyChangeEvent(event.items[i]) != null) {
             updateReservationBox(Reservation(PropertyChangeEvent(
              event.items[i]).source));
          }
        }
        else if (event.items[i] is Reservation) {
          updateReservationBox(Reservation(event.items[i]));
        }
    }
    break;

    case CollectionEventKind.RESET:
      refreshReservations();
      break;
  }
}
```

The `updateReservationBox()` method either shows or hides a box that shows the time span of the reservation. The `removeReservationBox()` method removes a reservation box. The `refreshReservations()` method redisplays all current reservation information.

For more information on the application and the individual methods, see the sample code.

Using collection change notifications

The IList and ICollectionView interfaces include the `itemUpdated()` method that notifies the collection that the underlying data has changed and ensures that the collection view of the data is up to date. This method takes the item that was modified, the property in the item that was updated, and its old and new values as parameters.

The ICollectionView interface also provides the `enableAutoUpdate()` and `disableAutoUpdate()` methods, which enable and disable the automatic updating of the collection view when the underlying data provider changes.

Using the itemUpdated method

Use the `itemUpdated()` method to notify the collection of changes to a data provider object if the object does not implement the IEventDispatcher interface; in this case the object is not monitorable. Flash and Flex Objects and other basic data types do not implement this interface. Therefore, You must use the `itemUpdated` method to update the collection when you modify the properties of a data provider such as an Array or through the display object.

You can also use the `itemUpdated()` method if you must use an Array, rather than a collection, as a control's data provider. Then the component wraps the Array in a collection wrapper. The wrapper needs to be manually notified of any changes made to the underlying Array data object, and you can use the `itemUpdated()` for that notification.

You do *not* have to use the `itemUpdated()` method if you add or remove items directly in a collection or use any of the ICollectionView or IList methods to modify the collection.

Also, specifying the `[Bindable]` metadata tag above a class definition, or above a variable declaration within the class ensures that the class implements the IEventDispatcher interface, and causes the class to dispatch propertyChange events. If you specify the `[Bindable]` tag above the class declaration, the class dispatches propertyChange events for all properties; if you mark only specific properties as `[Bindable]`, the class dispatches events for only those properties. The collection listens for the propertyChange events. Therefore, if you have a collection called `myCollection` that consists of instances of a class that has a `[Bindable] myVariable` variable, an expression such as `myCollection.getItemAt(0).myVariable="myText"` causes the item to dispatch an event and you do not have to use the `itemUpdated()` method. (For more information on the [Bindable] metadata tag and its use, see Chapter 38, "Binding Data," on page 1245.)

The most common use for the `itemUpdate()` method is to notify a collection of changes to a custom class data source that you cannot make [Bindable] or modify to implement the IEventDispatcher interface. The following schematic example shows how you could use the `itemUpdated()` method in such a circumstance:

Assume you have a class that you do not control or edit that looks like the following:

```
public class ClassICantEdit {
  public var field1:String;
  public var field2:String;
}
```

You have an ArrayCollection that uses these object, such as the following, which you populate with `classICantEdit` objects.

```
public var myCollection:ArrayCollection = new ArrayCollection();
```

You have DataGrid control such as the following

```
<mx:DataGrid dataProvider="{myCollection}"/>
```

When you update a field in the `myCollection` ArrayCollection, as follows, the DataGrid does not automatically update.

```
myCollection.getItemAt(0).field1="someOtherValue";
```

To update the DataGrid control, you must use the collection's `itemUpdated()` method:

```
myCollection.itemUpdated(collectionOfThoseClasses.getItemAt(0));
```

Disabling and enabling automatic updating

The ICollectionView `disableAutoUpdate()` method prevents events that represent changes to the underlying data from being broadcast by the view. It also prevents the ICollectionView from updating as a result of these changes.

Use this method to prevent the ICollectionView, and therefore the control that uses it as a DataProvider, from showing intermediate changes in a set of multiple changes. The DataGrid class, for example, uses the `disableAutoUpdate()` method to prevent updates to the ICollectionView object while a specific item is selected. When the item is no longer selected, the DataGrid calls the `enableAutoUpdate()` method. Doing this ensures that, if a DataGrid uses a sorted collection view, items that you edit do not jump around while you're editing.

You can also use the `disableAutoUpdate()` method to optimize performance in cases where multiple items in a collection are being edited at once. By disabling the auto update until all changes are made, a control like DataGrid can receive an update event as a single batch instead of reacting to multiple events.

The following code snippet shows the use of the `disableAutoUpdate()` and `enableAutoUpdate()` methods:

```
var obj:myObject = myCollection.getItemAt(0);
myCollection.disableAutoUpdate();
obj.prop1 = 'foo';
obj.prop2 = 'bar';
myCollection.enableAutoUpdate();
```

Example: Modifying data in DataGrid control

The following example lets you add, remove, or modify data in a DataGrid control.

```
<?xml version="1.0"?>
<!-- dpcontrols\ModifyDataGridData.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500"
    height="600" >

    <mx:Script>
        <![CDATA[
            import mx.events.*;
            import mx.collections.*;

            // Add event information to a log (displayed in the TextArea).
            public function
collectionEventHandler(event:CollectionEvent):void {
                switch(event.kind) {
                    case CollectionEventKind.ADD:
                        addLog("Item "+ event.location + " added");
                        break;
                    case CollectionEventKind.REMOVE:
                        addLog("Item "+ event.location + " removed");
                        break;
                    case CollectionEventKind.REPLACE:
                        addLog("Item "+ event.location + " Replaced");
                        break;
                    case CollectionEventKind.UPDATE:
                        addLog("Item updated");
                        break;
                }
            }
            // Helper function for adding information to the log.
            public function addLog(str:String):void {
                log.text += str + "\n";
            }

            // Add a person to the ArrayCollection.
            public function addPerson():void {
                ac.addItem({first:firstInput.text, last:lastInput.text,
                    email:emailInput.text});
                    clearInputs();
            }

            // Remove a person from the ArrayCollection.
            public function removePerson():void {
                // Make sure an item is selected.
                if (dg.selectedIndex >= 0) {
                    ac.removeItemAt(dg.selectedIndex);
            }
```

```
        }

        // Update an existing person in the ArrayCollection.
        public function updatePerson():void {
            // Make sure an item is selected.
            if (dg.selectedItem !== null) {
                ac.setItemAt({first:firstInput.text, last:lastInput.text,
                    email:emailInput.text}, dg.selectedIndex);
            }
        }

        // The change event listener for the DataGrid.
        // Clears the text input controls and updates them with the contents
        // of the selected item.
        public function dgChangeHandler():void {
            clearInputs();
            firstInput.text = dg.selectedItem.first;
            lastInput.text = dg.selectedItem.last;
            emailInput.text = dg.selectedItem.email;
        }

        // Clear the text from the input controls.
        public function clearInputs():void {
            firstInput.text = "";
            lastInput.text = "";
            emailInput.text = "";
        }

        // The labelFunction for the ComboBox;
        // Puts first and last names in the ComboBox.
        public function myLabelFunc(item:Object):String {
            return item.first + " " + item.last;
        }
        ]]>
    </mx:Script>

    <!-- The ArrayCollection used by the DataGrid and ComboBox. -->
    <mx:ArrayCollection id="ac"
            collectionChange="collectionEventHandler(event)">
        <mx:source>
            <mx:Object first="Matt" last="Matthews" email="matt@myco.com"/>
            <mx:Object first="Sue" last="Sanderson" email="sue@myco.com"/>
           <mx:Object first="Harry" last="Harrison" email="harry@myco.com"/
>
        </mx:source>
    </mx:ArrayCollection>

    <mx:DataGrid width="450" id="dg" dataProvider="{ac}"
            change="dgChangeHandler()">
        <mx:columns>
```

```
            <mx:DataGridColumn dataField="first" headerText="First Name"/>
            <mx:DataGridColumn dataField="last" headerText="Last Name"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>

    <!-- The ComboBox and DataGrid controls share an ArrayCollection as
their
        data provider.
        The ComboBox control uses the labelFunction property to construct
the
        labels from the dataProvider fields. -->
    <mx:ComboBox id="cb" dataProvider="{ac}" labelFunction="myLabelFunc"/>

    <!-- Form for data to add or change in the ArrayCollection. -->
    <mx:Form>
       <mx:FormItem label="First Name">
            <mx:TextInput id="firstInput"/>
       </mx:FormItem>
       <mx:FormItem label="Last Name">
            <mx:TextInput id="lastInput"/>
       </mx:FormItem>
       <mx:FormItem label="Email">
            <mx:TextInput id="emailInput"/>
       </mx:FormItem>
    </mx:Form>

    <mx:HBox>
        <!-- Buttons to initiate operations on the collection. -->
        <mx:Button label="Add New" click="addPerson()"/>
        <mx:Button label="Update Selected" click="updatePerson()"/>
        <mx:Button label="Remove Selected" click="removePerson()"/>
        <!-- Clear the text input fields. -->
        <mx:Button label="Clear" click="clearInputs()"/>
    </mx:HBox>

    <!-- The application displays event information here -->
    <mx:Label text="Log"/>
    <mx:TextArea id="log" width="100" height="100%"/>
</mx:Application>
```

Using hierarchical data providers

You use hierarchical data providers with the controls that display a nested hierarchy of nodes and subnodes, such as tree branches and leaves and Menu submenus and items. The following controls take hierarchical data providers:

- Menu
- MenuBar
- PopUpMenuButton
- Tree

The hierarchical components all use the same mechanism to work with the data provider. The examples in this section use the Tree control but also apply to the other components also.

About hierarchical data providers

The Flex framework, by default, supports two types of hierarchical data sources:

XML can be any of the following: Strings containing well-formed XML; or XML, XMLList, or XMLListCollection objects, including objects generated by the `<mx:XML>` and `<mx:XMLList>` compile-time tags. (These tags support data binding, which you cannot do directly in ActionScript.) Flex can automatically structure a Tree or menu-based control to reflect the nesting hierarchy of well-formed XML.

Objects can be any set of nested Objects or Object subclasses (including Arrays or ArrayCollection objects) having a structure where the children of a node are in a `children` field. For more information, see "Creating a custom data descriptor". You can also use the `<mx:Model>` compile-time tag to create nested objects that support data binding, but you must follow the structure defined in "Using the <mx:Model> tag with Tree and menu-based controls" on page 200.

You can add support for other hierarchical data provider structures, such as nested Objects where the children might be in fields with varying names.

Data descriptors and hierarchical data provider structure

Hierarchical data used in Tree and menu-based controls must be in a form that can be parsed and manipulated using a data descriptor class. A *data descriptor* is a class that provides an interface between the hierarchical control and the data provider object. It implements a set of control-specific methods to determine the data provider contents and structure, and to get, add, and remove data, and to change control-specific data properties.

Flex defines two data descriptor interfaces for hierarchical controls:

ITreeDataDescriptor Methods used by Tree controls

IMenuDataDescriptor Methods for Menu, MenuBar, and PopUpMenuButton controls

The Flex framework provides a DefaultDataDescriptor class that implements both interfaces. You can use the `dataDescriptor` property to specify a custom data descriptor class that handles data models that do not conform to the default descriptor structure.

Data descriptor methods and source requirements

The following table describes the methods of both interfaces, and the behavior of the DefaultDataDescriptor class. The first line of each Interface/Method entry indicates whether the method belongs to the ITreeDataDescriptor interface, IMenuDataDescriptor interface, or both interfaces, and therefore indicates whether the method is used for trees, menus, or both.

Interface/ Method	Returns	DefaultDataDescriptor behavior
Both hasChildrent(*node*, [*model*])	Boolean value indicating whether the node a branch with children.	For XML, returns `true` if the node has at least one child element. For other objects, returns `true` if the node has a non-empty `children` field.
Both getChildren(*node, [collection]*)	A node's children.	For XML, returns an XMLListCollection with the child elements. For other Objects, returns the contents of the node's `children` field.
Both isBranch(*node, [collection]*)	Whether a node is a branch.	For XML, returns true if the node has at least one child, or if it has an `isBranch` attribute. For other Objects, returns `true` if the node has an `isBranch` field.
Both getData(*node, [collection]*)	The node data.	Returns the node.
Both addChildAt(*node*, *child*, *index*, [*model*])	Boolean value indicating whether the operation succeeded.	For all cases, inserts the node as a child object before the node currently in the index location.
Both removeChildAt (*node*, *index*, [*model*])	Boolean value indicating whether the operation succeeded.	For all cases, removes the child of the node in the index location.

Interface/ Method	Returns	DefaultDataDescriptor behavior
IMenuDataDescriptor getType(*node*)	A String with the menu node type, Meaningful values are `check`, `radio`, and `separator`.	For XML, returns the value of the `type` attribute of the node. For other Objects, returns the contents of the node's `type` field.
IMenuDataDescriptor isEnabled(*node*)	Boolean value indicating if a menu node is enabled.	For XML, returns the value of the `enabled` attribute of the node. For other Objects, returns the contents of the node's `enabled` field.
IMenuDataDescriptor setEnabled(*node*, *value*)		For XML, sets the value of the `enabled` attribute of the node to `true` or `false`. For other Objects, sets the contents of the node's `enabled` field.
IMenuDataDescriptor isToggled(*node*)	Boolean value indicating if a menu node is selected	For XML, returns the value of the `selected` attribute of the node. For other Objects, returns the contents of the node's `enabled` field.
IMenuDataDescriptor setToggled(*node*, *value*)		For XML, sets the value of the `selected` attribute of the node to `true` or `false`. For other Objects, sets the contents of the node's `enabled` field.
IMenuDataDescriptor getGroupName (*node*)	The name of the radio button group to which the node belongs.	For XML, returns the value of the `groupName` attribute of the node. For other Objects, returns the contents of the node's `groupName` field.

The following example Object follows the default data provider structure for a Tree control, and is correctly handled by the DefaultDataDescriptor class:

```
[Bindable]
public var fileSystemStructure:Object =
  {label:"mx", children: [
    {label:"Containers", children: [
      {label:"Accordian", children:[]},
      {label:"DividedBox", children: [
        {label:"BoxDivider.as", data:"BoxDivider.as"},
        {label:"BoxUniter.as", data:"BoxUniter.as"}]},
      {label: "Grid", children:[]}]},
    {label: "Controls", children: [
      {label: "Alert", data: "Alert.as"},
      {label: "Styles", children: [
        {label: "AlertForm.as", data:"AlertForm.as"}]},
```

```
      {label: "Tree", data: "Tree.as"},
      {label: "Button", data: "Button.as"}]},
    {label: "Core", children:[]}
  ]};
```

For objects, the root is the Object instance, so there must always a single root (as with XML). You could also use an Array containing nested Arrays as the data provider; in this case the provider has no root; each element in the top level array appears at the top level of the control.

The DefaultDataDescriptor can properly handle well formed XML nodes. The `isBranch()` method, however, returns true only if the parameter node has child nodes or if the node has an `isBranch` attribute with the value `true`. If your XML object uses any technique other than a `true` `isBranch` attribute to indicate empty branches, you must therefore create a custom data descriptor.

The DefaultDataDescriptor handles collections properly. For example, if a node's children property is an ICollectionView, the `getChildren()` method returns the children as an ICollectionView object.

Using the <mx:Model> tag with Tree and menu-based controls

The `<mx:Model>` tag lets you define a data provider structure in MXML. The Flex compiler converts the contents of the tag into a hierarchical graph of ActionScript Objects. The `<mx:Model>` tag has two advantages over defining an Object data provider in ActionScript:

- You can define the structure using an easily read, XML-like format.
- You can bind structure entries to ActionScript variables, so you can use the `<mx:Model>` to create an object-based data provider that gets its data from multiple dynamic sources.

To use an `<mx:Model>` tag with a control that uses a data descriptor, the object generated by the compiler must conform to the data descriptor requirements, as discussed in "Data descriptors and hierarchical data provider structure" on page 197. Also, as with an XML object, the tag must have a single root element.

In most situations, you should consider using an `<mx:XML>` or `<mx:XMLList>` tag, as described in "Using an XML data provider" on page 209, instead of using an `<mx:Model>` tag. The XML-based tags support data binding to elements, and the DefaultDataDescriptor class supports all well-structured XML. Therefore, you can use a more natural structure, where node names can represent their function, and you do not have to artificially name nodes "children."

To use the an `<mx:Model>` tag as the data provider for a control that uses DefaultDataDescriptor class, all child nodes must be named "children." This requirement differs from the structure that you use with an Object, where the array that contains the child objects is named children.

The following example shows the use of an <mx:Model> tag with data binding as a data provider for a menu, and shows how you can change the menu structure dynamically:

```
<?xml version="1.0"?>
<!-- dpcontrols\ModelWithMenu.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*">
    <mx:Script>
        <![CDATA[

            import mx.controls.Menu;
            public var productMenu:Menu;

            public function initMenu(): void
            {
                productMenu = Menu.createMenu(null, Products.Department);
                productMenu.setStyle("disabledColor", 0xCC3366);
                productMenu.show(10,10);
            }

        ]]>
    </mx:Script>

    <mx:Model id="Products">
        <Root>
            <Department label="Toys">
                <children label="Care Bear"/>
                <children label="GI Joe"/>
                <children label="Telly Tubbies"/>
            </Department>
            <Department label="Kitchen">
                <children label="Electronics">
                    <children label="Crock Pot"/>
                    <children label="Panini Grill"/>
                </children>
                <children label="Cookware">
                    <children label="Grill Pan"/>
                    <children label="Iron Skillet" enabled="false"/>
                </children>
            </Department>
            <!-- The items in this entry are bound to the form data -->
            <Department label="{menuName.text}">
                <children label="{item1.text}"/>
                <children label="{item2.text}"/>
                <children label="{item3.text}"/>
            </Department>
        </Root>
    </mx:Model>

    <mx:Button label="Show Products" click="initMenu()"/>
    <!-- If you change the contents of the form, the next time you
```

```
        display the Menu, it will show the updated data in the last
        main menu item. -->
    <mx:Form>
        <mx:FormItem label="Third Submenu title">
            <mx:TextInput id="menuName" text="Clothing"/>
        </mx:FormItem>
        <mx:FormItem label="Item 1">
            <mx:TextInput id="item1" text="Sweaters"/>
        </mx:FormItem>
        <mx:FormItem label="Item 2">
            <mx:TextInput id="item2" text="Shoes"/>
        </mx:FormItem>
        <mx:FormItem label="Item 3">
            <mx:TextInput id="item3" text="Jackets"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Creating a custom data descriptor

If your hierarchical data does not fit the formats supported by the DefaultDataDescriptor class, for example if your data is in an object that does not use a children field, you can write a custom data descriptor and specify it in your Tree control's `dataDescriptor` property. The custom data descriptor must implement all methods of the `ITreeDataDescriptor` interface.

The following example shows how you can create a custom data descriptor, in this case, for use with a Tree control. This data descriptor correctly handles a data provider that consists of nested ArrayCollection objects.

The following code shows the MyCustomTreeDataDescriptor class, which implements only the ITreeDataDescriptor interface, so it supports Tree controls, but not menu-based controls. The custom class supports tree nodes whose children field is either an ArrayCollection or an Object. When getting a node's children, if the children object is an ArrayCollection, it returns the object; otherwise, it wraps the children object in an ArrayCollection before returning it. When adding a node, it uses a different method to add the node, depending on the children field type.

```
package myComponents
// myComponents/MyCustomTreeDataDescriptor.as
{
import mx.collections.ArrayCollection;
import mx.collections.CursorBookmark;
import mx.collections.ICollectionView;
import mx.collections.IViewCursor;
import mx.events.CollectionEvent;
import mx.events.CollectionEventKind;
import mx.controls.treeClasses.*;

public class MyCustomTreeDataDescriptor implements ITreeDataDescriptor
{

    // The getChildren method requires the node to be an Object
    // with a children field.
    // If the field contains an ArrayCollection, it returns the field
    // Otherwise, it wraps the field in an ArrayCollection.
    public function getChildren(node:Object,
        model:Object=null):ICollectionView
    {
        try
        {
            if (node is Object) {
                if(node.children is ArrayCollection){
                    return node.children;
                }else{
                    return new ArrayCollection(node.children);
                }
            }
        }
        catch (e:Error) {
            trace("[Descriptor] exception checking for getChildren");
        }
        return null;
    }

    // The isBranch method simply returns true if the node is an
    // Object with a children field.
    // It does not support empty branches, but does support null children
    // fields.
```

```
    public function isBranch(node:Object, model:Object=null):Boolean {
        try {
            if (node is Object) {
                if (node.children != null)  {
                    return true;
                }
            }
        }
        catch (e:Error) {
            trace("[Descriptor] exception checking for isBranch");
        }
        return false;
    }

    // The hasChildren method Returns true if the
    // node actually has children.
    public function hasChildren(node:Object, model:Object=null):Boolean {
        if (node == null)
            return false;
        var children:ICollectionView = getChildren(node, model);
        try {
            if (children.length > 0)
                return true;
        }
        catch (e:Error) {
        }
        return false;
    }
    // The getData method simply returns the node as an Object.
    public function getData(node:Object, model:Object=null):Object {
        try {
            return node;
        }
        catch (e:Error) {
        }
        return null;
    }

    // The addChildAt method does the following:
    // If the parent parameter is null or undefined, inserts
    // the child parameter as the first child of the model parameter.
    // If the parent parameter is an Object and has a children field,
    // adds the child parameter to it at the index parameter location.
    // It does not add a child to a terminal node if it does not have
    // a children field.
    public function addChildAt(parent:Object, child:Object, index:int,
            model:Object=null):Boolean {
        var event:CollectionEvent = new
CollectionEvent(CollectionEvent.COLLECTION_CHANGE);
        event.kind = CollectionEventKind.ADD;
```

```
        event.items = [child];
        event.location = index;
        if (!parent) {
            var iterator:IViewCursor = model.createCursor();
            iterator.seek(CursorBookmark.FIRST, index);
            iterator.insert(child);
        }
        else if (parent is Object) {
            if (parent.children != null) {
                if(parent.children is ArrayCollection) {
                    parent.children.addItemAt(child, index);
                    if (model){
                        model.dispatchEvent(event);
                        model.itemUpdated(parent);
                    }
                    return true;
                }
                else {
                    parent.children.splice(index, 0, child);
                    if (model)
                        model.dispatchEvent(event);
                    return true;
                }
            }
        }
        return false;
    }

    // The removeChildAt method does the following:
    // If the parent parameter is null or undefined,
    // removes the child at the specified index
    // in the model.
    // If the parent parameter is an Object and has a children field,
    // removes the child at the index parameter location in the parent.
    public function removeChildAt(parent:Object, child:Object, index:int,
model:Object=null):Boolean
    {
        var event:CollectionEvent = new
CollectionEvent(CollectionEvent.COLLECTION_CHANGE);
        event.kind = CollectionEventKind.REMOVE;
        event.items = [child];
        event.location = index;

        //handle top level where there is no parent
        if (!parent)
        {
            var iterator:IViewCursor = model.createCursor();
            iterator.seek(CursorBookmark.FIRST, index);
            iterator.remove();
            if (model)
```

```
                model.dispatchEvent(event);
            return true;
        }
        else if (parent is Object)
        {
            if (parent.children != undefined)
            {
                parent.children.splice(index, 1);
                if (model)
                    model.dispatchEvent(event);
                return true;
            }
        }
        return false;
    }

}
}
```

The following example uses the MyCustomTreeDataDescriptor to handle hierarchical nested ArrayCollections and objects. When you click the button it adds a node to the tree by calling the data descriptor's `addChildAt()` method. Notice that you would not normally use the `addChildAt()` method directly. Instead, you would use the methods of a Tree or menu-based control, which, in turn, use the data descriptor methods to modify the data provider.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- dpcontrols\CustDataDescriptor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
creationComplete="initCollections()">

    <mx:Script>
        <![CDATA[
            import mx.collections.*;
            import mx.controls.treeClasses.*;
            import myComponents.*;

            //Variables used to construct the ArrayCollection data provider
            //First top-level node and its children.
            public var nestArray1:Array = [
                {label:"item1", children: [
                    {label:"item1 child", children:     [
                        {label:"item 1 child child", data:"child data"}
                    ]}
                ]}
            ];
            //Second top-level node and its children.
            public var nestArray2:Array = [
                {label:"item2", children: [
                    {label:"item2 child", children: [
                        {label:"item 2 child child", data:"child data"}
                    ]}
                ]}
            ];
            //Second top-level node and its children.
            public var nestArray3:Array = [
                {label:"item3", children: [
                    {label:"item3 child", children: [
                        {label:"item 3 child child", data:"child data"}
                    ]}
                ]}
            ];
            //Variable for the tree array.
            public var treeArray:Array
            //Variables for the three Array collections that correspond to
the
            //top-level nodes.
            public var col1:ArrayCollection;
            public var col2:ArrayCollection;
```

```
            public var col3:ArrayCollection;
            //Variable for the ArrayCollection used as the Tree data
provider.
            [Bindable]
            public var ac:ArrayCollection;

            //build the ac ArrayCollection from its parts.
            public function initCollections():void{
                // Wrap each top-level node in an ArrayCollection.
                col1 = new ArrayCollection(nestArray1);
                col2 = new ArrayCollection(nestArray2);
                col3 = new ArrayCollection(nestArray3);
                // Put the three top-level node
                // ArrayCollections in the treeArray.
                treeArray = [
                    {label:"first thing", children: col1},
                    {label:"second thing", children: col2},
                    {label:"third thing", children: col3},
                ];
                //Wrap the treeArray in an ArrayCollection.
                ac = new ArrayCollection(treeArray);
            }

            // Adds a child node as the first child of the selected node,
            // if any. The default selectedItem is null, which causes the
            // data descriptor addChild method to add it as the first child
            // of the ac ArrayCollection.
            public function clickAddChildren():void {
                var newChild:Object = new Object();
                newChild.label = "New Child";
                newChild.children = new ArrayCollection();
               tree.dataDescriptor.addChildAt(tree.selectedItem, newChild,
0, ac);
            }

        ]]>
    </mx:Script>

    <mx:Tree width="200" id="tree" dataProvider="{ac}"
        dataDescriptor="{new MyCustomTreeDataDescriptor()}"/>
    <mx:Button label="add children" click="clickAddChildren()"/>
</mx:Application>
```

Using an XML data provider

Often, the data for a tree is retrieved from a server in the form of XML, but it can also be well-formed XML defined within the <mx:Tree> tag. The DefaultDataDescriptor class can handle well-formed XML data providers.

You can use an <mx:XML> or <mx:XMLList> tag to define an XML or XMLList object in MXML. Unlike the XML and XMLList classes in ActionScript, these tags let you use MXML binding expressions in the XML text to extract node contents from variable data. For example, you can bind a node's name attribute to a text input value, as in the following example:

```
<mx:XMLList id="myXMLList">
  <child name="{textInput1.text}"/>
  <child name="{textInput2.text}"/>
</mx:XMLList>
```

You can use an XML object directly as a dataProvider to a hierarchical data control. However, if the object changes dynamically, you should do the following:

1. Convert the XML or XMLList object to an XMLListCollection object.
2. Make all update to the data by modifying the XMLListCollection object.

Doing this ensures that the component represents the dynamic data. The XMLListCollection class supports the use of all IList and ICollectionView interface methods, and adds many of the most commonly used XMLList class methods. For more information on using XMLListCollections see "Using the XMLListCollection class" on page 211.

The following code example defines two Tree controls; the first uses an XML object directly, the second uses an XMLListCollection object as the data source:

```
<?xml version="1.0"?>
<!-- dpcontrols\UseXMLDP.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:XML id="capitals">
        <root>
            <Capitals label="U.S. State Capitals">
                <capital label="AL" value="Montgomery"/>
                <capital label="AK" value="Juneau"/>
                <capital label="AR" value="Little Rock"/>
                <capital label="AZ" value="Phoenix"/>
            </Capitals>
            <Capitals label="Canadian Province Capitals">
                <capital label="AB" value="Edmonton"/>
                <capital label="BC" value="Victoria"/>
                <capital label="MB" value="Winnipeg"/>
                <capital label="NB" value="Fredericton"/>
            </Capitals>
        </root>
    </mx:XML>

    <!-- Create an XMLListCollection representing the Tree nodes.
         capitals.Capitals is an XMLList with both Capitals elements. -->
    <mx:XMLListCollection id="capitalColl" source="{capitals.Capitals}"/>

    <!-- When you use an XML-based data provider with a tree
         you must specify the label field, even if it
         is "label". The XML object includes the root,
         so you must set showRoot="false". Remember that
         the Tree will not, by default, reflect dynamic changes
         to the XML object. -->
    <mx:Tree id="Tree1" dataProvider="{capitals}" labelField="@label"
        showRoot="false" width="300"/>
    <!-- The XMLListCollection does not include the XML root. -->
    <mx:Tree id="Tree2" dataProvider="{capitalColl}" labelField="@label"
        width="300"/>
</mx:Application>
```

This example shows two important features when using a hierarchical data provider with a Tree control:

- ECMAScript for XML (E4X) objects must have a single root node, which might not appropriate for displaying in the Tree. Also, trees, can have multiple elements at their highest level. To prevent the tree from displaying the root node, specify the `showRoot` property to `false`. (The default `showRoot` value for the Tree control is `true`.) XMLList collections, however, do not have a single root, and you typically do not need to use the `showRoot` property.
- When you use an XML, XMLList, or XMLListCollection object as the tree data provider, you must specify the `labelField` property, even if it is "label", if the field is an XML attribute. You must do this because you must use the @ sign to signify an attribute.

Using the XMLListCollection class

The XMLListCollection class provides collection functionality to an XMLList object and makes available some of the XML manipulation methods of the native XMLList class, such as the `attributes()`, `children()`, and `elements()`. For details of the supported methods, see XMLListCollection in *Adobe Flex 2 Language Reference*.

The following simple example uses an XMLListCollection object as the data provider for a List control. It uses XMLListCollection methods to dynamically add and remove items from the data provider and its representation in the List control. The example uses a Tree control to represent a selection of shopping items and a List collection to represent a shopping list.

Users add items to the List control by selecting an item in a Tree control (which uses a static XML object as its data provider) and clicking a button. When the user clicks the button, the event listener uses the XMLListCollection `addItem()` method to add the selected XML node to the XMLListCollection. Because the data provider is a collection, the List control updates to show the new data.

Users remove items in a similar manner, by selecting an item in the list and clicking a Remove button. The event listener uses the XMListCollection `removeItemAt()` method to remove the item from the data provider and its representation in the List control.

```
<?xml version="1.0"?>
<!-- dpcontrols\XMLListCollectionWithList.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="400">
    <mx:Script>
        <![CDATA[
            import mx.collections.XMLListCollection;
            import mx.collections.ArrayCollection;

            // An XML object with categorized produce.
            [Bindable]
            public var myData:XML=
                <catalog>
                  <category name="Meat">
                      <product name="Buffalo" cost="4" isOrganic="No"
                          isLowFat="Yes"/>
                      <product name="T Bone Steak" cost="6" isOrganic="No"
                          isLowFat="No"/>
                      <product name="Whole Chicken" cost="1.5"
isOrganic="Yes"
                           isLowFat="No"/>
                  </category>
                  <category name="Vegetables">
                      <product name="Broccoli" cost="2.16" isOrganic="Yes"
                          isLowFat="Yes"/>
                      <product name="Vine Ripened Tomatoes" cost="1.69"
isOrganic="No"
                          isLowFat="Yes"/>
                      <product name="Yellow Peppers" cost="1.25"
isOrganic="Yes"
                          isLowFat="Yes"/>
                  </category>
                  <category name="Fruit">
                      <product name="Bananas" cost="0.95" isOrganic="Yes"
                          isLowFat="Yes"/>
                      <product name="Grapes" cost="1.34" isOrganic="No"
                          isLowFat="Yes" />
                    <product name="Strawberries" cost="2.5" isOrganic="Yes"
                          isLowFat="Yes"/>
                  </category>
              </catalog>;
            // An XMLListCollection representing the data
            // for the shopping List.
           [Bindable]
           public var listDP:XMLListCollection = new XMLListCollection(new
XMLList());
```

```
            // Add the item selected in the Tree to the List XMLList data
provider.
            private function doTreeSelect():void
            {
                if (prodTree.selectedItem)
                listDP.addItem(prodTree.selectedItem.copy());
            }
           // Remove the selected in the List from the XMLList data provider.
            private function doListRemove():void
            {
                if (prodList.selectedItem)
                    listDP.removeItemAt(prodList.selectedIndex);
            }
        ]]>
    </mx:Script>

    <mx:Tree id="prodTree" dataProvider="{myData}" width="200"
        showRoot="false" labelField="@name"/>

    <mx:HBox>
        <mx:Button id="treeSelect" label="Add to List"
            click="doTreeSelect()"/>
        <mx:Button id="listRemove" label="Remove from List"
            click="doListRemove()"/>
    </mx:HBox>

    <mx:List id="prodList" dataProvider="{listDP}" width="200"
        labelField="@name"/>
</mx:Application>
```

Using remote data providers

You can use the following types of remote data providers for Flex components:

- RPC data sources: HTTPService components, WebService components, and RemoteObject components
- Flex DataService components that you use in conjunction with the server-side Flex Data Management Service to distribute and synchronize data among multiple client applications.

The following sections describe how you can use these remote data sources to provide data. For more information on using remote data providers see Chapter 45, “Using RPC Components,” on page 1407.

Using an RPC data source

To use an RPC data source, you represent the result of the remote service by using the appropriate class as follows:

- The RemoteObject class automatically returns as an ArrayCollection any data that is represented on the server as a java.util.List object, and you can use the returned object directly.
- Otherwise, including for HTTPService and WebService results, convert the return data to a collection class if the data changes or if you use the same result in multiple places (in the latter case, you increase efficiency). As a general rule, use an ArrayCollection for serialized (list-based) objects or an XMLListCollection for data in E4X format.

The following code excerpt shows this use, converting a list returned by a web service to an ArrayCollection:

```
  <mx:WebService id="employeeWS" destination"employeeWS"
    showBusyCursor="true"
    fault="alert(event.fault.faultstring)">
    <mx:operation name="getList">
      <mx:request>
        <deptId>{dept.selectedItem.data}</deptId>
      </mx:request>
    </mx:operation>
.
.
  </mx:WebService>

  <mx:ArrayCollection id="ac"
    source="mx.utils.ArrayUtil.toArray(employeeWS.getList.lastResult)"/>
  <mx:DataGrid dataProvider="{ac}" width="100%">
```

For more information on using RPC data sources, see Chapter 45, “Using RPC Components,” on page 1407.

Using a DataService component

To use a DataService component as a data provider, you call the component’s `fill()` method to fill an ArrayCollection object with the data from a Data Management Service destination, as the following code excerpt shows:

```
<mx:Script>
  <![CDATA[
    import mx.data.DataService;
    import mx.collections.ArrayCollection;

    public var ds:DataService;
    [Bindable]
```

```
    public var contacts:ArrayCollection;

    public function initApp()
    {
      contacts = new ArrayCollection();
      ds = new DataService("contact");
      ds.fill(contacts);
    }
  ]]>
</mx:Script>
.
.
    <mx:DataGrid id="dg" dataProvider="{contacts}" editable="true">
```

For more information on using DataService components, see Chapter 51, "Distributing Data in Flex Applications," on page 1501.

Using paged remote data providers.

When you use a DataService class to get your remote data, you can have a collection that does not initially load all of its data on the client. By using this technique, you can prevent large amounts of data from traveling over the network and slowing down your application while that data is processed. The data that you get incrementally is referred to as *paged* data, and the data that has not yet been received is *pending* data.

About ItemPendingError errors

When you retrieve paged data, your code might try to access pending data. In this case, the collection throws an ItemPendingError error.

Several Flex controls automatically catch and handle ItemPendingError errors thrown by the dataProvider collection so that your application does not have to manage the errors. These controls include: List, HorizontalList, TileList, DataGrid, Menu, and Tree.

Most other classes, including all chart controls, do not handle item pending errors, and you must write your own error handling code.

You must handle ItemPendingError errors in the following cases:

- **A control that does not handle ItemPendingError errors** If the control uses a paged collection as its data provider, you need to ensure that the data is fully loaded before passing it to the control, or your application must watch for unexpected behavior from the control.

- **You write code that uses a paged collection directly** Consider whether your code will ever attempt to handle pending data. If so, wrap the code that accesses the collection data in try/catch blocks for the ItemPendingError. For example, if your code iterates through an entire collection, wrap the `while` loop in a try/catch block and continue calling until the iteration is complete.

Handling ItemPendingError errors

All classes that implement the ICollectionView and IList interfaces throw an ItemPendingError when Flex attempts to access paged collection data that is not yet available. The ItemPendingError class provides a techniques for finding out about the status of the requested data as follows:

- The ItemPendingError object has an `addResponder()` method that lets you specify an array of one or more IResponder objects. Flex framework provides an ItemResponder class that implements the IResponder interface, or you can create your own implementation class.
- Each IResponder object must have two functions, a *result* function that Flex calls when the data is successfully retrieved and a *fault* function that Flex calls if the retrieval fails. You code the functions to handle these circumstances as needed by your application. The ItemResponder class constructor takes these two functions as parameters, and takes a third, optional Object parameter that the two functions can use in their processing.

To handle an ItemPendingError:

1. Put the code that might generate the error in a `try` block.
2. Immediately follow the `try` block with a `catch` block with the following signature:

```
catch (e:ItemPendingError) {
```

3. Create one or more new responder objects, each with the following format:

```
responder1 = new ItemResponder(
  // The result function
  function (data:Object, token:Object=null) {
    // Code to handle newly received data goes here.
    }
  // The fault function
    function (info:Object, token:Object=null) {
    // Code to handle a failure where data cannot become available
    // goes here.
    }
    // The function must take an optional Object parameter; for
    // information see ItemResponder in ActionScript 3.0 Language
  Reference.
);
```

4. Add the responder objects to the ItemPendingError object, as follows:

```
e.addResponder(responder1);
```

The two functions that you pass in to the Responder constructor define the IResponder method implementations: the first function defines the implementation of the IResponder `fault` method, and the second function defines the implementation of the IResponder `result` method. The preceding example uses members of the Flex ItemResponder class as the responder objects, so it defines methods that take a second, optional parameter.

The following code shows how a function that iterates over a paged collection can handle ItemPendingError errors:

```
private var myCursor:IViewCursor;
private var total:int = 0;

// Iterate over the myView collection
private function iterate():void
{
  if (myCursor == null)
      myCursor = myView.createCursor();
  // Put the code that moves the cursor in a try block.
  try
  {
    while(!myCursor.afterLast)
    {
      total += myCursor.current.amount;
      myCursor.moveNext();
    }
    trace('Total amount is:', total);
  }
  // The catch block handles the error generated when the requested data
  // is pending.
  catch (e:ItemPendingError)
  {
    // Create a new Responder object and assign it to the error object's
    // responder property.
    responder = new ItemResponder(
      // Define a function to handle case where the data becomes available.
      function (data:Object, token:Object=null) {
        myCursor.moveNext();
        iterate();
      },
      // Define a function to handle case where a "real" error occurs.
      function (info:Object, token:Object=null) {
        trace('fault when retrieving data', info.toString());
      });
  }
}
```

ItemPendingError notes

Almost all cursor functions may throw ItemPendingErrors. It is worth noting that if an ItemPendingError is thrown the cursor will remain at its last known good value, meaning that if you call moveNext() and the error is thrown the cursor will have remained on the old value of current. A smaller number of methods will throw an error on IList (getItemAt and getIndexOf primarily), check the ASDoc for details.

Data might not always be loaded sequentially, so it is possible that you will be in the middle of an ICollectionView and will movePrevious with your cursor and will encounter an ItemPendingError.

Ensuring all data is available before you display a control

If your application uses a control, such as a chart control, that requires the data provider to contain the complete data set, your application must ensure that all the data in a collection is available before assigning the control's `dataProvider` property. In most cases, you can do this by configuring the DataService not to use paging. However, in some cases, such as those where you use the same Data Management Service destination for multiple purposes in your Flex application, you might need to use paged data in your control.

One technique for using paged data in a control that requires complete data is to use the IList interface `toArray()` method. This method attempts to load all the data in its parameter object into an Array. If not all the data is available, it throws an ItemPendingError error, and you can handle the error as described in "Handling ItemPendingError errors" on page 216. Alternatively, you can iterate from the beginning to the end of the data before using it in the control.

The following example shows this use to ensure that a paged data provider (specified by the `theDP` variable) has complete data before using it in a chart control:

```
private function loadDP():void
{
  var cursor:IViewCursor = theDP.createCursor();
  try
  {
    while (cursor.moveNext()) {}
    chart.dataProvider = theDP;
  }
  catch (e:ItemPendingError)
  {
    e.addResponder(new ItemResponder(
      function (result:Object, token:Object=null)
      {
        loadDP();
      },
```

```
        function (fault:Object, token:Object=null)
        {
          trace('Error while loading');
        }
      ));
      cursor = null; //Might as well let it garbage collect.
    }
}
```

CHAPTER 8

Sizing and Positioning Components

8

Adobe Flex lays out components by determining their sizes and positions; it provides you with multiple options for determining both sizes and positions. This topic discusses how Flex lays out components, and how you can control component size and position.

Contents

About sizing and positioning

Flex controls the layout of components by using a set of rules. The layout rules are a combination of sizing rules for individual components, and sizing and positioning rules for containers. Flex supports automatic layout, so you often do not have to initially set the size or position of components. Instead, you can concentrate on building the logic of your application and let Flex control the layout. Later, you can adjust the dimensions of instances, if necessary.

Each container has its own rules for controlling layout. For example, the VBox container lays out its children in a single column. A Grid container lays out its children in rows and columns of cells. The Application container has 24-pixel padding, and many other containers have 0-pixel padding.

Although Flex has built-in default layout rules, you can use the component's properties and methods to customize the layout. All components have several properties, including `height` and `width`, for specifying the component's size in absolute or container-relative terms. Each container also has properties and styles that you can use to configure aspects of layout. You can use settings such as the `verticalGap` and `horizontalGap` styles of a Tile container to set the spacing between children, and the `direction` property to specify a row or column layout. You can also use different positioning techniques for laying out components in a container; some containers, for example, support absolute *x*- and *y*-coordinate–based positioning.

About layout in Flex

The Layout Manager controls layout in Flex. The manager uses the following three-stage process to determine the size and position of each component in an application:

1. **Commitment pass** Determines the property settings of the application's components. This phase allows components whose contents depend on property settings to configure themselves before Flex determines their sizes and positions.

   During the commitment pass, the Layout Manager causes each component to run its `commitProperties()` method, which determines the property values.

2. **Measurement pass** Calculates the default size of every component in the application. This pass starts from the most deeply nested components and works out toward the Application container. The measurement pass determines the *measured*, or default, size of each component. The default size of each container is based on the default or explicit (if specified) sizes of its children. For example, the Box container's default width is equal to the sum of the default or explicit widths of all of its children, plus the thickness of the borders, plus the padding, plus the gaps between the children.

   During the measurement pass, the Layout Manager causes each component to run its `measureSizes()` method, which calls the `measure()` method, to determine the component's default size.

3. **Layout pass** Lays out your application, including moving and resizing any components. This pass starts from the outermost container and works in toward the innermost component. The layout pass determines the actual size and placement of each component. It also does any programmatic drawing, such as calls to the `lineTo()` or `drawRect()` methods.

During the layout pass, Flex determines whether any component's sizing properties specify dimensions that are a percentage of the parent, and uses the setting to determine the child component's actual size. The Layout Manager causes each component to run its `updateDisplayList()` method to lay out the component's children; for this reason, this pass is also referred to as the update pass.

About Flex frames of reference

Flex uses several frames of reference in determining positions and sizes:

- The local area and local coordinate system are relative to the outer edges of the component. The component's visual elements, such as borders and scroll bars, are included in the local coordinates.
- The viewable area is the region of a component that is inside the component's visual elements; that is, it is the part of the component that is being displayed and can contain child controls, text, images, or other contents. Flex does not have a separate coordinate system for this area.
- The content area and content coordinate system include *all* of the component's contents, and do not include the visual elements. It includes any regions that are currently clipped from view and must be accessed by scrolling the component. The content area of a scrolling TextArea control, for example, includes the region of text that is currently scrolled off the screen.

Flex uses the viewable area when it determines percentage-based sizes and when it performs constraint-based layout.

Flex component `x` and `y` properties, which you use to specify absolute positioning, are in the content coordinate system.

NOTE	Flex coordinates increase from the upper left corner of the frame of reference. Thus, an *x,y* position of 100,300 in the local coordinate system is 100 pixels is to the right and 300 pixels down from the component's upper left corner.

For more information on Flex coordinate systems, see "Using Flex coordinates" on page 510.

About component sizing

The measurement and layout passes determine a component's height and width. You can get these dimensions by using the `height` and `width` properties; you can use these properties and others to control the component's size.

Flex provides several ways for you to control the size of controls and containers:

Default sizing Flex automatically determines the sizes of controls and containers.

Explicit sizing You set the `height` and `width` properties to absolute values.

Percentage-based sizing You specify the component size as a percentage of its container size.

Constraint-based layout You control size and position by anchoring component's sides to locations in their container.

For details on controlling component sizes, see "Sizing components" on page 228.

About component positioning

Flex positions components when your application initializes. Flex also performs a layout pass and positions or repositions components when the application or a user does something that could affect the sizes or positions of visual elements, such as the following situations:

- The application changes properties that specify sizing, such as `x`, `y`, `width`, `height`, `scaleX`, and `scaleY`.
- A change affects the calculated width or height of a component, such as when the label text for a Button control changes, or the user resizes a component.
- A child is added or removed from a container, a child is resized, or a child is moved. For example, if your application can change the size of a component, Flex updates the layout of the container to reposition its children, based on the new size of the child.
- A property or style that requires measurement and drawing, such as `horizontalScrollPolicy` or `fontFamily`, changes.

There are very few situations where an application programmer must force the layout of a component; for more information, see "Manually forcing layout" on page 227.

Flex provides two mechanisms for positioning and laying out controls:

Automatic positioning Flex automatically positions a container's children according to a set of container- and component-specific rules. Most containers, such as Box, Grid, or Form, use automatic positioning. Automatic positioning is sometimes referred to as automatic layout.

Absolute positioning You specify each child's, `x` and `y` properties, or use a constraint-based layout that specifies the distance between one or more of the container's sides and the child's sides or center. Absolute positioning is sometimes referred to as absolute layout.

Three containers support absolute positioning:

- The Application and Panel containers use automatic positioning by default, and absolute positioning if you specify the `layout` property as `"absolute"`.
- The Canvas container always uses absolute positioning.

For details on controlling the positions of controls, see "Positioning and laying out controls" on page 248.

Component layout patterns

Flex uses different patterns to lay out different containers and their children. These patterns generally fit in the type categories listed in the following table. The table describes the general layout behavior for each type, how the default size of the container is determined, and how Flex sizes percentage-based children.

Container type	Default layout behavior
Absolute positioning: Canvas, container or Application or Panel container with `layout="absolute"`	**General layout:** Children of the container do not interact. That is, children can overlap and the position of one child does not affect the position of any other child. You specify the child positions explicitly or use constraints to anchor the sides or centers of the children relative to the parent container. **Default sizing:** The measurement pass finds the child with the lowest bottom edge and the child with the rightmost edge, and uses these values to determine the container size. **Percentage-based children:** Sizing uses different rules depending on whether you use constraint-based layout or *x*- and *y*- coordinate positioning. See "Sizing percentage-based children of a container with absolute positioning" on page 239.
Controls that arrange all children linearly, such as Box, HBox, VBox	**General layout:** All children of the container are arranged in a single row or column. Each child's height and width can differ from all other children's heights or widths. **Default sizing:** The container fits the default or explicit sizes of all children and all gaps, borders, and padding. **Percentage based children:** If children with percentage-based sizing request more than the available space, the actual sizes are set to fit in the space, proportionate to the requested percentages.

Container type	Default layout behavior
Grid	**General layout:** The container is effectively a VBox control with rows of HBox child controls, where all items are constrained to align with each other. The heights of all the cells in a single row are the same, but each row can have a different height. The widths of all cells in a single column are the same, but each column can have a different width. You can define a different number of cells for each row or each column of the Grid container, and individual cells can span columns or rows. **Default sizing:** The grid fits the individual rows and children at their default sizes. **Percentage-based children:** If children use percentage-based sizing, the sizing rules fit the children GridItem components within their rows, and GridRow components within the grid size according to linear container sizing rules.
Tile	**General layout:** The container is a grid of equal-sized cells. The cells can be in row-first or column-first order. If you do not specify explicit or percentage-based dimensions, the control has as close as possible to an equal number of rows and columns, with the `direction` property determining the orientation with the larger number of items, if necessary. **Default sizing:** If you do not specify `tileWidth` and `tileHeight` properties, the container uses the measured or explicit size of the largest child cell for the size of each child cell. **Percentage based children:** The percentage-based sizes of a child component specify a percentage of the individual cell, not of the Tile container.
Navigators: ViewStack, Accordion, TabNavigator	**General layout:** The container displays one child at a time. **Default sizing:** The container size is determined by the measured or explicit dimensions of the initially selected child, and thereafter, all children are forced to be that initial size. If you set the `resizeToChild` property to `true` the container resizes to accommodate the measured or explicit size of each child, as that child appears. **Percentage based children:** As a general rule, you either use 100% for both height and width, which causes the children to fill the navigator bounds, or do not use percentage-based sizing.

Basic layout rules and considerations

Flex performs layout according to the following basic rules. If you remember these rules, you should be able to easily understand the details of Flex layout. These rules should help you determine why Flex lays out your application as it does and to determine how to modify your application appearance.

For a detailed description of how Flex sizes components, see "Determining and controlling component sizes" on page 231. For detailed information on component positioning, see "Positioning and laying out controls" on page 248.

- Flex first determines all components' measured (default) or explicitly-set sizes *up*, from the innermost child controls to the outermost (Application) control. This is done in the measurement pass.
- After the measurement pass, Flex determines all percentage-based sizes and lays out components *down*, from the outermost container to the innermost controls. This is done in the layout pass.
- Sizes that you set to a pixel value are mandatory and fixed, and override any maximum or minimum size specifications that you set for the component.
- The default sizes determined in the measurement pass specify the sizes of components that do not have explicit or percentage-based sizes (or use constraint-based layout), and are fixed.
- Percentage-based size specifications are advisory. The layout algorithms satisfy the request if possible, and use the percentage values to determine proportional sizes, but the actual sizes can be less than the requested sizes. Percentage-based sizes are always within the component's maximum and minimum sizes, and, subject to those bounds, don't cause a container's children to exceed the container size.

Manually forcing layout

Sometimes, you must programmatically cause Flex to lay out components again. Flex normally delays processing properties that require substantial computation until the script that causes them to be set finishes executing. For example, setting the `width` property is delayed, because it may require recalculating the widths of the object's children or its parent. Delaying processing prevents it from being repeated multiple times if the script sets the object's `width` property more than once. However, in some situations, you might have to force the layout before the script completes.

Situations where you must force a layout include the following circumstances:

- When printing multiple page data grids by using the PrintDataGrid class.
- Before playing an effect, if the start values have just been set on the target.
- When capturing bitmap data after making property changes.

To force a layout, call the `validateNow()` method of the component that needs to be laid out. This method causes Flex to validate and update the properties, sizes, and layout of the object and all its children, and to redraw them, if necessary. Because this method is computation-intensive, you should be careful to call it only when it is necessary.

For an example of using the `validateNow()` method, see "Updating the PrintDataGrid layout" on page 1174.

Sizing components

Flex provides several ways for controlling the size of components. You can do the following:

- Have Flex automatically determine and use default component sizes.
- Specify pixel sizes.
- Specify component size as a percentage of the parent container.
- Combine layout and sizing by specifying a constraint-based layout.

The following sections describe the basic sizing properties, provide details on how Flex determines component sizes, describe how to use automatic, explicit, and percentage-based sizing, and describe various techniques for controlling component size. For information on constraint-based layout, see "Using constraint-based layout" on page 255.

Flex sizing properties

Several Flex properties affect the size of components. As a general rule, you use only a few properties for most applications, but a more complete understanding of these properties can help you understand the underlying Flex sizing mechanism and how Flex sizing properties interrelate. For more information on Flex component sizes, see "Determining and controlling component sizes" on page 231.

The following sections describe the most commonly used sizing properties, followed by tables that describe all characteristics and properties that determine how Flex sizes components. The tables include properties that are not normally used by many application developers; these properties can be used by developers of custom components, particularly, those who must implement a custom `measure()` method. The last subsection describes specific sizing behaviors and techniques that can be useful in developing applications.

Commonly used sizing properties

If you are not creating custom components, you typically use the following basic properties to specify how a component is sized:

- The `height`, `width`, `percentHeight`, and `percentWidth` properties specify the height and width of a component. In MXML tags, you use the `height` and `width` properties to specify the dimensions in pixels or as percentages of the parent container size. In ActionScript, you use the `height` and `width` properties to specify the dimensions in pixels, and use the `percentHeight` and `percentWidth` properties to specify the dimensions as a percentage of the parent container.
- The `minHeight`, `minWidth`, `maxHeight`, and `maxWidth` properties specify the minimum and maximum dimensions that a component can have if Flex determines the component size. These properties have no effect if you explicitly set the width or height in pixels.

The following sections describe these properties in greater detail and explain how they relate to each other. Flex provides a number of powerful and subtle sizing features, and understanding these concepts can help you, even if you never use any properties other than the ones listed in this section. The section "Determining and controlling component sizes" on page 231 describes the rules that Flex applies to determine the sizes of components based on the properties.

Basic sizing characteristics and properties

The following characteristics and their associated properties determine the size of the component:

Characteristic	Associated properties	Description
Actual dimensions	Returned by the `height` and `width` properties.	The height and width of the displayed control, in pixels, as determined by the layout phase. If you set any explicit values, they determine the corresponding actual values.
Explicit dimensions	`explicitHeight`, `explicitWidth` Setting the `height` and `width` properties to integer values also sets the `explicitHeight` and `explicitWidth` properties.	A dimension that you specifically set as a number of pixels. These dimensions cannot be overridden. Application developers typically use the `height` and `width` properties to set explicit dimensions. You cannot have both an explicit dimension and a percentage-based dimension, setting one unsets the other.

Characteristic	Associated properties	Description
Percentage-based dimensions	`percentHeight`, `percentWidth` In MXML tags only, setting the `height` and `width` properties to percentage string values, such as "50%" also sets the `percentHeight` and `percentWidth` properties.	A dimension that you specifically set as a number in the range 0-100, as a percentage of the viewable area of the parent container. If you set a percentage-based dimension, the component is resizable, and grows or shrinks if the parent dimension changes.
Default dimensions	`measuredHeight`, `measuredWidth`	Not used directly by application developers. The dimensions of the component, as determined by the `measure()` method of the component. These values cannot be outside the range determined by the component's maximum and minimum height and width values. For more information on maximum and minimum default sizes, see "Maximum and minimum dimensions".

Maximum and minimum dimensions

The following characteristics determine the minimum and maximum *default* and *percentage-based* dimensions that a component can have. They *do not* affect values that you set explicitly or dimensions determined by using constraint-based layout.

Characteristic	Associated Properties	Description
Minimum dimensions	`minHeight`, `minWidth` Setting the explicit minimum dimensions also sets the. `minHeight` and `minWidth` properties.	The minimum dimensions a component can have. By default, Flex sets these dimensions to the values of the minimum default dimensions.

Characteristic	Associated Properties	Description
Maximum dimensions	`maxHeight`, `maxWidth` Setting the explicit maximum dimensions also sets the `maxHeight` and `maxWidth` properties.	The maximum dimensions a component can have. The default values of these properties are component-specific, but often are 10000 pixels.
Minimum default dimensions	`measuredMinHeight`, `measuredMinWidth`	Not used by application developers. The minimum valid dimensions, as determined by the `measure()` method. The default values for these properties are component-specific; for many controls, the default values are 0.

Determining and controlling component sizes

The following sections describe in detail how Flex determines the sizes of controls and containers based on the components and their properties and how you can use Flex properties to control the sizes. The first section describes how you can use different sizing properties to control component size. The second section describes general sizing rules that apply to both controls and containers. The third section describes rules for sizing containers.

NOTE

For a summary of the basic rules for component sizing, see "Basic layout rules and considerations" on page 226.

Basic sizing property rules

The following rules describe how you can use Flex sizing properties to specify the size of a component:

- Any dimension property that you set overrides the corresponding default value; for example, an explicitly set `height` property overrides any default height.
- Setting the `width`, `height`, `maxWidth`, `maxHeight`, `minWidth`, or `minHeight` property to a pixel value in MXML or ActionScript also sets the corresponding explicit property, such as `explicitHeight` or `explicitMinHeight`.
- The explicit height and width and the percentage-based height and width are mutually exclusive. Setting one value sets the other to `NaN`; for example, if you set `height` or `explicitHeight` to 50 and then set `percentHeight` to 33, the value of the `explicitHeight` property is `NaN`, not 50, and the `height` property returns a value that is determined by the `percentHeight` setting.

- If you set the `height` or `width` property to a percentage value in an MXML tag, you actually set the percentage-based value, that is, the `percentHeight` or `percentWidth` property, *not* the explicit value. In ActionScript, you cannot set the `height` or `width` property to a percentage value; instead, you must set the `percentHeight` or `percentWidth` property.
- When you get the `height` and `width` properties, the value is always the actual height or width of the control.

Determining component size

During the measurement pass, Flex determines the components' default, (also called measured) sizes. During the layout pass, Flex determines the actual sizes of the components, based on the explicit or default sizes and any percentage-based size specifications.

The following list describes sizing rules and behaviors that apply to all components, including both controls and containers. For container-specific sizing rules, see "Determining container size" on page 233. For detailed information on percentage-based sizing, see "Using percentage-based sizing" on page 237.

- If you specify an explicit size for any component (that is not outside the component's minimum or maximum bounds), Flex always uses that size.
- If you specify a percentage-based size for any component, Flex determines the component's actual size as part of the parent container's sizing procedure, based on the parent's size, the component's requested percentage, and the container-specific sizing and layout rules.
- The default and percentage-based sizes are always at least as large as any minimum size specifications.
- If you specify a component size by using a percentage value and do not specify an explicit or percentage-based size for its container, the component size is the default size. Flex ignores the percentage specification. (Otherwise, an infinite recursion might result.)
- If a child or set of children require more space than is available in the parent container, the parent clips the children at the parent's boundaries, and, by default, displays scroll bars on the container so users can scroll to the clipped content. Set the `clipContent` property to `false` to configure a parent to let the child extend past the parent's boundaries. Use the `scrollPolicy` property to control the display of the scroll bars.
- If you specify a percentage-based size for a component Flex uses the viewable area of the container in determining the sizes.

- When sizing and positioning components, Flex does not distinguish between visible and invisible components. By default, an invisible component is sized and positioned as if it were visible. To prevent Flex from considering an invisible component when it sizes and positions other components, set the component's `includeInLayout` property to `false`. This property affects the layout of the children of all containers except Accordion, FormItem, or ViewStack. For information on using the "Preventing layout of hidden controls" on page 251.

NOTE

Setting a component's `includeInLayout` property to `false` does not prevent Flex from laying out or displaying the component; it only prevents Flex from considering the component when it lays out other components. As a result, the next component or components in the display list overlap the component. To prevent Flex from displaying the component, also set the `visible` property to `false`.

Determining container size

Flex uses the following basic rules, in addition to the basic component sizing rules, to determine the size of a container:

- Flex determines all components' default dimensions during the measurement pass, and uses these values when it calculates container size during the layout pass.
- If you specify an explicit size for a container, Flex always uses that size, as with any component.
- If you specify a percentage-based size for a container, Flex determines the container's actual size as part of the parent container's sizing procedure, as with any component.
- A percentage-based container size is advisory. Flex makes the container large enough to fit its children at their minimum sizes. For more information on percentage-based sizing, see "Using percentage-based sizing" on page 237.
- If you do not specify an explicit or percentage-based size for a container, Flex determines the container size by using explicit sizes that you specify for any of its children, and the default sizes for all other children.
- Flex does not consider any percentage-based settings of a container's children when sizing the container; instead, it uses the child's default size.
- If a container uses automatic scroll bars, Flex does not consider the size of the scroll bars when it determines the container's default size in its measurement pass. Thus, if a scroll bar is required, a default-sized container might be too small for proper appearance.

Each container has a set of rules that determines the container's default size. For information on default sizes of each control and container, see the specific container sections in *Adobe Flex 2 Language Reference*, and in the following topics in *Flex 2 Developer's Guide*: Chapter 14, "Using the Application Container," on page 529; Chapter 15, "Using Layout Containers," on page 553; and Chapter 16, "Using Navigator Containers," on page 627.

Example: Determining an HBox container and child sizes

The following example code shows how Flex determines the sizes of an HBox container and its children. In this example, the width of the HBox container is the sum of the default width of the first and third buttons, the minimum width of the second button (because the default width would be smaller), and 16 for the two gaps. The default width for buttons is based on the label text width; in this example it is 66 pixels for all three buttons. The HBox width, therefore, is 66 + 70 + 66 + 16 = 218. If you change the `minWidth` property of the second button to 50, the calculation uses the button's default width, 66, so the HBox width is 214.

When Flex lays out the application, it sets the first and third button widths to the default values, 66, and the second button size to the minimum width, 70. It ignores the percentage-based specifications when calculating the final layout.

```
<?xml version="1.0"?>
<!-- components\HBoxSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:HBox id="h21">
        <mx:Button id="bG1"
            label="Label 1"
            width="50%"/>
        <mx:Button id="bG2"
            label="Label 2"
            width="40%"
            minWidth="70"/>
        <mx:Button id="bG3"
            label="Label 3"/>
    </mx:HBox>

    <mx:TextArea height="50" width="100%">
        <mx:text>
            HBox: {h21.width}  Button1: {bG1.width}  Button2: {bG2.width}
                Button3: {bG3.width}
        </mx:text>
    </mx:TextArea>
</mx:Application>
```

In the following example, the HBox width now is 276 pixels, 50% of 552 pixels, where 552 is the Application container width of 600 minus 48 pixels for the 24-pixel left and right container padding. The button sizes are 106, 85, and 66 pixels respectively. The third button uses the default size. The variable width button sizes are five-ninths and four-ninths of the remaining available space after deducting the default-width button and the gaps, and the 1-pixel-wide border.

If you set the HBox `width` property to 20%, however, the HBox width is *not* 120 pixels, 20% of the Application container width, because the this value is too small to fit the HBox container's children. Instead it is 200, the sum of 66 pixels (the default size) for buttons 1 and 3, 50 pixels (the specified minimum size) for button 2, 16 pixels for the gaps between buttons, and 2 pixels for the border. The buttons are 66, 50, and 66 pixels wide, respectively.

```
<?xml version="1.0"?>
<!-- components\HBoxSizePercent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox id="h21" width="50%" borderStyle="solid">
        <mx:Button id="bG1"
            label="Label 1"
            width="50%"/>
        <mx:Button id="bG2"
            label="Label 2"
            width="40%"
            minWidth="50"/>
        <mx:Button id="bG3"
            label="Label 3"/>
    </mx:HBox>

    <mx:TextArea height="50" width="100%">
        <mx:text>
            HBox: {h21.width}  Button1: {bG1.width}  Button2: {bG2.width}
                Button3: {bG3.width}
        </mx:text>
    </mx:TextArea>
</mx:Application>
```

For more information and examples showing sizing of containers and children, see "Using Flex component sizing techniques" on page 236. For detailed information on percentage-based sizing, see "Using percentage-based sizing" on page 237.

Using Flex component sizing techniques

The following sections describe briefly how you can use default sizing, explicit sizing, and percentage-based sizing techniques to control the size of components. For information on using constraint-based layout for component sizing, see "Using constraint-based layout" on page 255.

Using default sizing

If you do not otherwise specify sizes, the component's `measure()` method calculates a size based on the default sizing characteristics of the particular component and the default or explicit sizes of the component's child controls.

As a general rule, you should determine whether a component's default size (as listed for the component in *Flex 2 Developer's Guide*) is appropriate for your application. If it is, you do not have to specify an explicit or percentage-based size.

The following example shows how you can use default sizing for Button children of an HBox container. In this example, none of the children of the HBox container specify a width value:

```
<?xml version="1.0"?>
<!-- components\DefaultButtonSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400" borderStyle="solid">
        <mx:Button label="Label 1"/>
        <mx:Button label="Label 2"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

Flex, therefore, uses the default sizes of the buttons, which accommodate the button label and default padding, and draws this application as the following image shows:

Notice the empty space to the right of the third button, because the sum of the default sizes is less than the available space.

Specifying an explicit size

You use the `width` and `height` properties of a component to explicitly set its size, as follows:

```
<?xml version="1.0"?>
<!-- components\ExplicitTextSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox id="myHBox">
        <mx:TextInput id="myInput"
            width="200"
            height="40"/>
    </mx:HBox>
</mx:Application>
```

In this example, Flex sets the component sizes to 200 by 40 pixels.

The following example shows setting the sizes of a container and its child:

```
<?xml version="1.0"?>
<!-- components\ExplicitHBoxSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox id="myHBox" width="150" height="150">
        <mx:TextInput id="myInput"
            text="Enter the zip code"
            width="200"
            height="40"/>
    </mx:HBox>
</mx:Application>
```

Because the specified TextInput control size is larger than that of its parent HBox container, Flex clips the TextInput control at the container boundary and displays a scroll bar (if you do not disable it) so that you can scroll the container to the clipped content. For more information on scroll bar sizing considerations, see "Dealing with components that exceed their container size" on page 243.

Using percentage-based sizing

Percentage-based sizing dynamically determines and maintains a component's size relative to its container; for example, you can specify that the component's width is 75% of the container. This sizing technique has several advantages over default or explicit fixed sizing:

- You only have to specify a size relative to the container; you don't have to determine exact measurements.
- The component size changes dynamically when the container size changes.
- The sizing mechanism automatically takes into account the remaining available space and fits components even if their requested size exceeds the space.

To specify a percentage value, use one of the following coding techniques:

- In an MXML tag, set the `height` or `width` property to a percentage value; for example:
  ```
  <mx:TextArea id="ta1" width="70%" height="40%"/>
  ```
- In an MXML tag or an ActionScript statement, set the `percentHeight` or `percentWidth` property to a numeric value; for example:
  ```
  ta1.percentWidth=70;
  ```

The exact techniques Flex uses to determine the dimensions of a component that uses percentage-based sizing depend on the type of container that holds the container. For example, a Tile container has cells that are all the largest default or explicit dimensions of the largest child. Child control percentage values specify a percentage of the tile cell size, not of the Tile control size. The percentage sizes of the Box, HBox, and VBox containers, on the other hand, are relative to the container size.

Sizing percentage-based children of a linear container with automatic positioning

When Flex sizes children of a container that uses automatic positioning to lay out children in a single direction, such as a HBox or VBox container, Flex does the following:

1. Determines the size of the viewable area of the parent container, and uses the corresponding dimensions as the container dimensions for sizing calculations. The viewable area is the part of the component that is being displayed and can contain child controls, text, images, or other contents. For more information on calculating the size of containers, see "Determining component size" on page 232.
2. Determines the desired sizes of children with percentage-based sizes by multiplying the decimal value by the size of the viewable area of the container, minus any padding and inter-child gaps.
3. Reserves space for all children with explicit or default sizes.
4. If available space (parent container size minus all reserved space, including borders, padding, and gaps) cannot accommodate the percentage requests, divides the available space in proportion to the specified percentages.
5. If a minimum or maximum height or width specification conflicts with a calculated value, uses the minimum or maximum value, and recalculates all other percentage-based components based on the reduced available space.
6. Rounds the size down to the next integer.

The following examples show how the requested percentage can differ from the size when the component is laid out:

- Suppose that 50% of a HBox parent is available after reserving space for all explicit-sized and default-sized components, and for all gaps and padding. If one component requests 20% of the parent, and another component requests 60%, the first component is sized to 12.5% ((20 / 20+ 60) * 50%) of the parent container, the second component is sized to 37.5% of the parent container.
- If any component, for example, a Tile container, requests 100% of its parent Application container's space, it occupies all of the container *except* for the Application's 24-pixel-wide top, bottom, left, and right padding, unless you explicitly change the padding settings of the Application container.

Sizing percentage-based children of a container with absolute positioning

When Flex sizes children of a container that uses absolute positioning, it does the following:

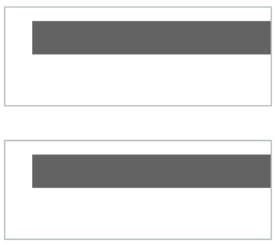
1. Determines the viewable area of the parent container, and uses the corresponding dimensions as the container dimensions for sizing calculations. For more information on calculating the size of containers, see "Determining component size" on page 232.
2. Determines the sizes of children with percentage-based sizes by multiplying the decimal value by the container dimension minus the position of the control in the dimension's direction. For example, if you specify `x="10"` and `width="100%"` for a child, the child size extends only to the edge of the viewable area, not beyond.

   Because controls can overlay other controls or padding, the sizing calculations do not consider padding or any other children when determining the size of a child.
3. If a minimum or maximum height or width specification conflicts with a calculated value, uses the minimum or maximum value.
4. Rounds the size down to the next integer.

The following code shows the percentage-based sizing behavior with absolute positioning:

```
<?xml version="1.0"?>
<!-- components\PercentSizeAbsPosit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]"
    verticalGap="25">

    <mx:Canvas
        width="200" height="75"
        borderStyle="solid">

        <mx:HBox
            x="20" y="10"
            width="100%" height="25"
            backgroundColor="#666666"/>
    </mx:Canvas>

    <mx:Canvas
        width="200" height="75"
        borderStyle="solid">

        <mx:HBox
            left="20" top="10"
            width="100%" height="25"
            backgroundColor="#666666"/>
    </mx:Canvas>
</mx:Application>
```

Flex draws the following application:



Examples: Using percentage-based children of an HBox container

The following example specifies percentage-based sizes for the first two of three buttons in an HBox container:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildren.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1" width="25%"/>
        <mx:Button label="Label 2" width="40%"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

In this example, the default width of the third button is 66 pixels. The HBox container has no padding by default, but it does put a 8-pixel horizontal gap between each component. Because this application has three components, these gaps use 16 pixels, so the available space is 384. The first button requests 25% of the available space, or 96 pixels. The second button requests 40% of 384 pixels, rounded down to 153 pixels. There is still unused space to the right of the third button.

Flex draws the following application:

Now change the percentage values requested to 50% and 40%, respectively:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildren5040.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1"
            width="50%"/>
        <mx:Button label="Label 2"
            width="40%"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

In this example, the first button requests 50% of the available HBox space, or 192 pixels. The second button still requests 40%, or 153 pixels, for a total of 345 pixels. However, the HBox only has 318 pixels free after reserving 66 pixels for the default-width button and 16 pixels for the gaps between components. Flex divides the available space proportionally between the two buttons, giving .5/(.5 + .4) * 318 = 176 pixels, to the first button and .4/(.5 + .4) * 318 = 141 pixels, to the second button. (All calculated values are rounded down to the nearest pixel.)

Flex draws the following application:

Using minimum or maximum dimensions

You can also use the `minWidth`, `minHeight`, `maxWidth`, and `maxHeight` properties with a percentage-based component to constrain its size. Consider the following example:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildrenMin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1"
            width="50%"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

To determine the widths of the percentage-based button sizes, Flex first determines the sizes as described in the second example in "Examples: Using percentage-based children of an HBox container" on page 241, which results in requested values of 176 for the first button and 141 for the second button. However, the minimum width of the second button is 150, so Flex sets its size to 150 pixels, and reduces the size of the first button to occupy the remaining available space, which results in a width of 168 pixels.

Flex draws the following application:

Sizing containers and components toolbox

The following sections describe specific techniques for controlling sizing, including setting the Application container size, handling components that exceed the container size, and using padding and custom gaps.

Setting the Application container size

When you size an application, you often start by setting the size of the Application container. The Application container determines the boundaries of your application in the Adobe Flash Player 9.

If you are using Flex Builder, or are compiling your MXML application on the server, an HTML wrapper page is generated automatically. The `width` and `height` properties specified in the `<mx:Application>` tag are used to set the width and height of the `<object>` and `<embed>` tags in the HTML wrapper page. Those numbers determine the portion of the HTML page that is allocated to the Flash plug-in.

If you are not autogenerating the HTML wrapper, set the `<mx:Application>` tag's `width` and `height` properties to 100%. That way, the Flex application scales to fit the space that is allocated to the Flash plug-in.

You set the Application container size by using the `<mx:Application>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\AppExplicit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="100"
    width="150">

        <!-- Application children go here. -->
</mx:Application>
```

In this example, you set the Application container size to 100 by 150 pixels. Anything in the application larger than this window is clipped at the window boundaries. Therefore, if you define a 200 pixel by 200 pixel DataGrid control, it is clipped, and the Application container displays scroll bars. (You can disable, or always display, scroll bars by setting the container's `horizontalScrollPolicy` and `verticalScrollPolicy` properties.)

For more information on sizing the Application container, see Chapter 14, "Using the Application Container," on page 529.

Dealing with components that exceed their container size

If the sum of the actual sizes of a container's children, plus the gaps and padding, exceed the dimensions of the container, by default, the container's contents are clipped at the container boundaries, and Flex displays scroll bars on the container so you can scroll to the remaining content. If you set the `horizontalScrollPolicy` and `verticalScrollPolicy` properties to `ScrollPolicy.OFF`, the scroll bars do not appear, but users do not have access to the clipped contents. If you set the `clipContent` property to `false`, container content can extend beyond the container boundary.

Using Scroll bars

If Flex cannot fit all of the components into the container, it uses scroll bars, unless you disable them by setting the `horizontalScrollPolicy` or `verticalScrollPolicy` property to `ScrollPolicy.OFF` or by setting `clipContent` to `false`. Consider the following example:

```
<?xml version="1.0"?>
<!-- components\ScrollHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1"
            width="50%"
            minWidth="200"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

In this example, the default width of the fixed-size button is 66 pixels, so there are 324 pixels of space available for the percentage-based buttons after accounting for the gap between components. The minimum widths of the first and second buttons are greater than the percentage-based values, so Flex assigns those buttons the set widths of 200 and 150 pixels, even though the HBox container only had 324 pixels free. The HBox container uses scroll bars to provide access to its contents because they now consume more space than the container itself.

Notice that the addition of the scroll bar doesn't increase the height of the container from its initial value. Flex considers scroll bars in its sizing calculations only if you explicitly set the scroll policy to `ScrollPolicy.ON`. So, if you use an auto scroll policy (the default), the scroll bar overlaps the buttons. To prevent this behavior, you can set the `height` property for the HBox container or allow the HBox container to resize by setting a percentage-based width. Remember that changing the height of the HBox container causes other components in your application to move and resize according to their own sizing rules. The following example adds an explicit height and permits you to see the buttons and the scroll bar:

```
<?xml version="1.0"?>
<!-- components\ScrollHBoxExplicitHeight.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400" height="42">
        <mx:Button label="Label 1"
            width="50%"
            minWidth="200"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

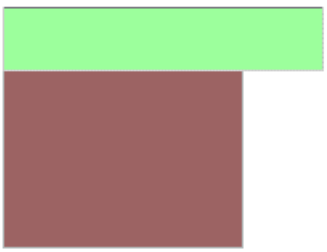
Alternately, you can set the HBox control's `horizontalScrollPolicy` property to `ScrollPolicy.ON`. This reserves space for the scroll bar during the initial layout pass, so it fits without overlapping the buttons or setting an explicit height. This also correctly handles the situation where the scroll bars change their size when you change skinning or styles. This technique places an empty scroll bar area on the container if it does not need scrolling, however.

Using the clipContent property

If you set the `clipContent` property for the parent container to `false`, the content can extend beyond the container's boundaries and no scroll bars appear, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\ClipHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600"
    height="400"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">

    <mx:HBox id="myHBox"
        width="150"
        height="150"
        borderStyle="solid"
        backgroundColor="#996666"
        clipContent="false">

        <mx:TextInput id="myInput"
            width="200" height="40"
            backgroundColor="#99FF99"/>
    </mx:HBox>
</mx:Application>
```

The following image shows the application, with the TextInput control extending past the right edge of the HBox control:



To ensure that components fit in the container, reduce the sizes of the child components. You can do this by setting explicit sizes that fit in the container, or by specifying percentage-based sizes. If you set percentage-based sizes, Flex shrinks the children to fit the space, or their minimum sizes, whichever is larger. By default, Flex sets the minimum height and width of most components to 0. You can set these components' minimum properties to nonzero values to ensure that they remain readable.

Using padding and custom gaps

There may be situations where you want your containers to have padding around the edges. (Previous Flex releases used the term margins; Flex 2 uses the term *padding* for consistency with cascading style sheet conventions.) Some containers, such as the Application container, have padding by default; others, such as the HBox container, have padding values of 0 by default. Also, some containers have gaps between children, which you might want to change from the default values. If your application has nonzero padding and gaps, Flex reserves the necessary pixels before it sizes any percentage-based components. Consider the following example:

```
<?xml version="1.0"?>
<!-- components\PadHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox
        width="400"
        borderStyle="solid"
        paddingLeft="5"
        paddingRight="5"
        horizontalGap="5">

        <mx:Button label="Label 1"
            width="50%"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

The default width of the fixed-size button is 66 pixels. All horizontal padding and gaps in the HBox control are 5 pixels wide, so the Flex application reserves 5 pixels for the left padding, 5 pixels for the right padding, and 10 pixels total for the two gaps between components, which leaves 314 pixels free for the two percentage-based components. Flex reserves 66 pixels for the default-sized (third) button; the second button requires its minimum size, 150 pixels; and the padding and gap take 20 pixels; this leaves 164 pixels available for the first button. The first button requests 200 pixels; therefore, it uses all available pixels and is 164 pixels wide.

Flex draws the following application:

Positioning and laying out controls

By default, Flex automatically positions all components, except for the children of a Canvas container. If you have a Canvas container, or an Application or Panel container with the `layout` property set to `absolute`, you specify absolute positions for its children, or use constraint-based layout. The following sections describe how to use automatic positions and absolute positioning by using `x` and `y` properties. For information on using constraint-based layout, which can control both positioning and sizing, see "Using constraint-based layout" on page 255.

Using automatic positioning

For most containers, Flex automatically positions the container children according to the container's layout rules, such as the layout direction, the container padding, and the gaps between children of that container.

For containers that use automatic positioning, setting the `x` or `y` property directly or calling `move()` has no effect, or only a temporary effect, because the layout calculations set the *x* position to the calculation result, not the specified value. You can, however, specify absolute positions for the children of these containers under some circumstances; for more information, see "Disabling automatic positioning temporarily" on page 249.

You can control aspects of the layout by specifying container properties; for details on the properties, see the property descriptions for the container in *Adobe Flex 2 Language Reference*. You also control the layout by controlling component sizes and by using techniques such as adding spacers.

The following sections describe specific techniques for controlling automatic positioning:

- Using the Spacer control to control layout
- Disabling automatic positioning temporarily
- Preventing layout of hidden controls

Using the Spacer control to control layout

Flex includes a Spacer control that helps you lay out children within a parent container. The Spacer control is invisible, but it does allocate space within its parent.

In the following example, you use a percentage-based Spacer control to push the Button control to the right so that it is aligned with the right edge of the HBox container:

```
<?xml version="1.0"?>
<!-- components\SpacerHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Image source="assets/flexlogo.jpg"/>
        <mx:Label text="Company XYZ"/>
        <mx:Spacer width="100%"/>
        <mx:Button label="Close"/>
    </mx:HBox>
</mx:Application>
```

In this example, the Spacer control is the only percentage-based component in the HBox container. Flex sizes the Spacer control to occupy all available space in the HBox container that is not required for other components. By expanding the Spacer control, Flex pushes the Button control to the right edge of the container.

You can use all sizing and positioning properties with the Spacer control, such as `width`, `height`, `maxWidth`, `maxHeight`, `minWidth`, and `minHeight`.

Disabling automatic positioning temporarily

You can use effects, such as the Move and Zoom effects, to modify the size or position of a child in response to a user action. For example, you might define a child so that when the user selects it, the child moves to the top of the container and doubles in size. These effects modify the `x` and `y` properties of the child as part of the effect. Similarly, you might want to change the position of a control by changing its *x* or *y* coordinate value, for example, in response to a button click.

Containers that use automatic positioning ignore the values of the `x` and `y` properties of their children during a layout update. Therefore, the layout update cancels any modifications to the `x` and `y` properties performed by the effect, and the child does not remain in its new location.

You can prevent Flex from performing automatic positioning updates that conflict with the requested action of your application by setting the `autoLayout` property of a container to `false`. Setting this property to `false` prevents Flex from laying out the container's contents when a child moves or resizes. Flex defines the `autoLayout` property in the Container class, and all containers inherit it; its default value is `true`, which enables Flex to update layouts.

Even when you set the `autoLayout` property of a container to `false`, Flex updates the layout when you add or remove a child. Application initialization, deferred instantiation, and the `<mx:Repeater>` tag add or remove children, so layout updates always occur during these processes, regardless of the value of the `autoLayout` property. Therefore, during container initialization, Flex defines the initial layout of the container children regardless of the value of the `autoLayout` property.

The following example disables layout updates for a VBox container:

```
<?xml version="1.0"?>
<!-- components\DisableVBoxLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox autoLayout="false"
        width="200"
        height="200">

        <mx:Button/>
        <mx:Button id="btn"
            click="btn.x += 10;"/>
        <mx:Button id="btn2"
            creationComplete="btn2.x = 100; btn2.y = 75;"/>
    </mx:VBox>
</mx:Application>
```

In this example, Flex initially lays out all three Button controls according to the rules of the VBox container. The creationComplete event listener for the third button is dispatched after the VBox control has laid out its children, but before Flex displays the buttons. Therefore, when the third button appears, it is at the *x* and *y* positions specified by the creationComplete listener. After the buttons appear, Flex shifts the second button 10 pixels to the right each time a user clicks it.

Setting the `autoLayout` property of a container to `false` prohibits Flex from updating a container's layout after a child moves or resizes, so you should set it to `false` only when absolutely necessary. You should always test your application with the `autoLayout` property set to the default value of `true`, and set it to `false` only as necessary for the specific container and specific actions of the children in that container.

For more information on effects, see Chapter 17, "Using Behaviors," on page 649.

Preventing layout of hidden controls

By default, Flex lays out and reserves space for all components, including hidden components, but it does not display the hidden controls. You see blank spots where the hidden controls will appear when you make them visible. In place of the hidden controls, you see their container's background. However if the container is any of the following components, you can prevent Flex from considering the child component when it lays out the container's other children by setting the child component's `includeInLayout` property of the component to `false`:

- Box, or any of its subclasses: HBox, VBox, DividedBox, HDividedBox, VdividedBox, Grid, GridItem, GridRow, ControlBar, and ApplicationControlBar,
- Form
- Tile and its subclass, Legend
- ToolBar

When a component's `includeInLayout` property is `false`, Flex does not include it in the layout calculations for other components, but still lays it out. In other words, Flex does not reserve space for the component, but still draws it. As a result, the component can appear underneath the components that follow it in the layout order. To prevent Flex from drawing the component, you must also set its `visible` property to `false`.

The following example shows the effects of the `includeInLayout` and `visible` properties. It lets you toggle each of these properties independently on the middle of three Panel controls in a VBox control.

```
<?xml version="1.0"?>
<!-- components\HiddenBoxLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <mx:Panel id="p1"
            title="Panel 1"
            backgroundColor="#FF0000"/>
        <mx:Panel id="p2"
            title="Panel 2"
            backgroundColor="#00FF00"/>
        <mx:Panel id="p3"
            title="Panel 3"
            backgroundColor="#0000FF"/>
    </mx:VBox>

    <mx:HBox>
        <mx:Button label="Toggle Panel 2 Visible"
            click="{p2.visible=!p2.visible;}"/>
        <mx:Button label="Toggle Panel 2 in Layout"
            click="{p2.includeInLayout=!p2.includeInLayout;}"/>
    </mx:HBox>
</mx:Application>
```

Run this application and click the buttons to see the results of different combinations of `visible` and `includeInLayout` properties. The example shows the following behaviors:

- If you include the second Panel control in the layout and make it invisible, Flex reserves space for it; you see the background of its VBox container in its place.
- If you do not include the second Panel control in the layout, the VBox resizes and the HBox with the buttons moves up. If you then include it in the layout, the VBox resizes again, and the HBox and buttons move down.
- If you do not include the second Panel control in the layout and make it visible, Flex still draws it, but does not consider it in laying out the third Panel control, so the two panels overlap. Because the title of a Panel control has a default alpha of 0.5, you see the combination of the second and third Panel controls in the second Panel position.

Using absolute positioning

Three containers support absolute positioning:

- Application and Panel controls use absolute positioning if you specify the `layout` property as `"absolute"` (`ContainerLayout.ABSOLUTE`).
- The Canvas container always uses absolute positioning.

With absolute positioning, you specify the child control position by using its `x` and `y` properties, or you specify a constraint-based layout; otherwise, Flex places the child at position 0,0 of the parent container. When you specify the *x* and *y* coordinates, Flex repositions the controls only when you change the property values. The following example uses absolute positioning to place a VBox control inside a Canvas control:

```
<?xml version="1.0"?>
<!-- components\CanvasLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">

    <mx:Canvas
        width="100" height="100"
        backgroundColor="#999999">

        <mx:VBox id="b1"
            width="80" height="80"
            x="20" y="20"
            backgroundColor="#A9C0E7">
        </mx:VBox>
    </mx:Canvas>
</mx:Application>
```

This example produces the following image:

When you use absolute positioning, you have full control over the locations of the container's children. This lets you overlap components. The following example adds a second VBox to the previous example so that it partially overlaps the initial box.

```
<?xml version="1.0"?>
<!-- components\CanvasLayoutOverlap.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">

    <mx:Canvas
        width="100" height="100"
        backgroundColor="#999999">

        <mx:VBox id="b1"
            width="80" height="80"
            x="20" y="20"
            backgroundColor="#A9C0E7">
        </mx:VBox>

        <mx:VBox id="b2"
            width="50" height="50"
            x="0" y="50"
            backgroundColor="#FF0000">
        </mx:VBox>
    </mx:Canvas>
</mx:Application>
```

This example produces the following image:

NOTE If you use percentage-based sizing for the children of a control that uses absolute positioning, be aware that percentage-based components resize when the parent container resizes, and the result may include unwanted overlapping of controls.

Using constraint-based layout

You can manage child component size and position simultaneously by using constraint-based layout, where you anchor the sides or center of a component to positions relative to the viewable region of the component's container. The viewable region is the part of the component that is being displayed, and it can contain child controls, text, images, or other contents.

NOTE

For an introduction to using constraint-based layout in Flex Builder, see *Getting Started with Flex 2*.

You can use constraint-based layout to determine the position and size of the immediate children of any container that supports absolute positioning.

With constraint-based layout you can do the following:

- Anchor one or more edges of a component at a pixel offset from the corresponding edge of its container's viewable region. The anchored child edge stays at the same distance from the parent edge when the container resizes. If you anchor both edges in a dimension, such as top and bottom, the component resizes if the container resizes.
- Anchor the child's horizontal or vertical center (or both) at a pixel offset from the center of the container's viewable region. The child does not resize in the specified dimension unless you also use percentage-based sizing.

Creating a constraint-based layout

The following rules specify how to position and size components by using constraint-based layout:

- Place the component directly inside a Canvas container, or directly inside an Application or Panel container with the `layout` property set to `absolute`.
- You can specify a constraint-based layout for any Flex framework component (that is, any component that extends the UIComponent class).
- Specify the constraints by using the `top`, `bottom`, `left`, `right`, `horizontalCenter`, or `verticalCenter` styles.

  The `top`, `bottom`, `left`, and `right` styles specify the distances between the component sides and the corresponding container sides.

  The `horizontalCenter` and `verticalCenter` styles specify distance between the component's center point and the container's center, in the specified direction; a negative number moves the component left or up from the center.

The following example anchors the Form control's left and right sides 20 pixels from its container's sides:

```
<mx:Form id="myForm" left="20" right="20"/>
```

- Do not specify a `top` or `bottom` style with a `verticalCenter` style; the `verticalCenter` value overrides the other properties. Similarly, do not specify a `left` or `right` style with a `horizontalCenter` style.
- A size determined by constraint-based layout overrides any explicit or percentage-based size specifications. If you specify `left` and `right` constraints, for example, the resulting constraint-based width overrides any width set by a `width` or `percentWidth` property.

Example: Using constraint-based layout for a form

The following example code shows how you can use constraint-based layout for a form. In this example, the Form control uses a constraint-based layout to position its top just inside the canvas padding. The form left and right edges are 20 pixels from the Canvas container's left and right edges. The HBox that contains the buttons uses a constraint-based layout to place itself 20 pixels from the Canvas right edge and 10 pixels from the Canvas bottom edge.

If you change the size of your browser or standalone Flash Player, you can see the effects of dynamically resizing the Application container on the Form layout. The form and the buttons overlap as the application grows smaller, for example. In an application, you should include the buttons in the last FormItem of the form, the buttons are separate in the following example to better show the effects of resizing.

```
<?xml version="1.0"?>
<!-- components\ConstraintLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Use a Canvas container in the Application to prevent
        unnecessary scroll bars on the Application. -->
    <mx:Canvas width="100%" height="100%">

         <!-- Anchor the top of the form at the top of the canvas.
            Anchor the form sides 20 pixels from the canvas sides. -->
        <mx:Form id="myForm"
                backgroundColor="#DDDDDD"
                top="0"
                left="20"
                right="20">

            <mx:FormItem label="Product:" width="100%">
               <!-- Specify a fixed width to keep the ComboBox control from
                    resizing as you change the application size. -->
                <mx:ComboBox width="200"/>
            </mx:FormItem>

            <mx:FormItem label="User" width="100%">
                <mx:ComboBox width="200"/>
            </mx:FormItem>

            <mx:FormItem label="Date">
                <mx:DateField/>
            </mx:FormItem>

            <mx:FormItem width="100%"
                    direction="horizontal"
                    label="Hours:">
                <mx:TextInput width="75"/>
                <mx:Label text="Minutes" width="48"/>
                <mx:TextInput width="75"/>
            </mx:FormItem>
        </mx:Form>

        <!-- Anchor the box with the buttons 20 pixels from the canvas
            right edge and 10 pixels from the bottom. -->
        <mx:HBox id="okCancelBox"
                right="20"
                bottom="10">
            <mx:Button label="OK"/>
            <mx:Button label="Cancel"/>
        </mx:HBox>
    </mx:Canvas>
</mx:Application>
```

CHAPTER 9

Using Controls

9

Controls are user-interface components such as Button, TextArea, and ComboBox controls. This topic describes how to use controls in a Flex application.

Adobe Flex has two types of controls: basic and data provider. This topic contains an overview of all Flex controls, and describes the basic Flex controls. For information on data provider controls, see Chapter 12, "Using Data-Driven Controls," on page 439.

Contents

About controls

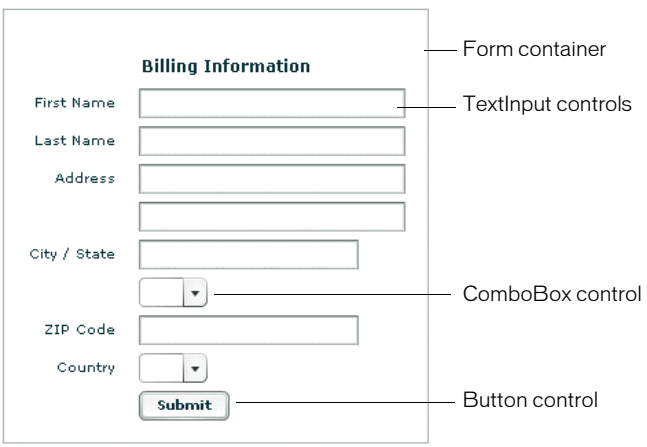
Controls are user-interface components, such as Button, TextArea, and ComboBox controls. You place controls in containers, which are user-interface components that provide a hierarchical structure for controls and other containers. Typically, you define a container, and then insert controls or other containers in it.

At the root of a Flex application is the `<mx:Application>` tag, which represents a base container that covers the entire Flash Player drawing surface. You can place controls or containers directly under the `<mx:Application>` tag or in other containers. For more information on containers, see Chapter 13, "Introducing Containers," on page 491.

Most controls have the following characteristics:

- MXML API for declaring the control and the values of its properties and events
- ActionScript API for calling the control's methods and setting its properties at run time
- Customizable appearance using styles, skins, and fonts

The following image shows several controls used in a Form container:



The MXML and ActionScript APIs let you create and configure a control. The following MXML code example creates a TextInput control in a Form container:

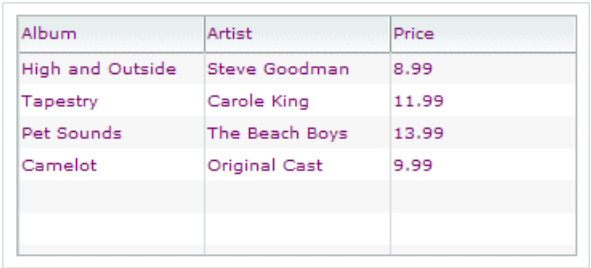
```
<?xml version="1.0"?>
<!-- controls\TextInputInForm.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form width="300" height="100">
        <mx:FormItem label="Card Name">
            <mx:TextInput id="cardName"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

This example produces the following image:

Although you commonly use MXML as the language for building Flex applications, you can also use ActionScript to configure controls. For example, the following code example populates a DataGrid control by providing an Array of items as the value of the DataGrid control's `dataProvider` property:

```
<?xml version="1.0"?>
<!-- controls\DataGridConfigAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

        private function myGrid_initialize():void {
          myGrid.dataProvider = [
            {Artist:'Steve Goodman', Album:'High and Outside', Price:8.99},
            {Artist:'Carole King', Album:'Tapestry', Price:11.99},
            {Artist:'The Beach Boys', Album:'Pet Sounds', Price:13.99},
            {Artist:'Original Cast', Album:'Camelot', Price:9.99} ];
        }
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        width="350" height="150"
        color="#7B0974"
        creationComplete="myGrid_initialize();"/>
</mx:Application>
```

This example produces the following image:



Album	Artist	Price
High and Outside	Steve Goodman	8.99
Tapestry	Carole King	11.99
Pet Sounds	The Beach Boys	13.99
Camelot	Original Cast	9.99

About text controls

Several Flex components display text or take text input, as the following table shows:

Control	Type of text
Label	Noneditable, single-line text field
Text	Noneditable, multiline text field
TextInput	(Optional) Editable, single-line text field
TextArea	(Optional) Editable, multiline text field
RichTextEditor	Compound control that contains a multiline text field and controls that let a user format text by selecting such characteristics as font, size, weight, alignment, and so on

These controls can display plain text that all has the same appearance. The controls can also display rich text formatted by using a subset of the standard HTML formatting tags. For information on using text controls, see Chapter 10, "Using Text Controls," on page 369.

Using data provider controls

Several Flex components, such as the DataGrid, Tree, and ComboBox controls, take input data from a data provider. A *data provider* is a collection of objects, similar to an array. For example, a Tree control reads data from the data provider to define the structure of the tree and any associated data assigned to each tree node.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same data provider, switch data providers for a component at run time, and modify the data provider so that changes are reflected by all components that use the data provider.

Consider that the data provider is the model, and the Flex components are the view onto the model. By separating the model from the view, you can change one without changing the other.

This topic describes the basic controls. For information on data provider controls, see Chapter 12, "Using Data-Driven Controls," on page 439.

Using menu controls

Several Flex controls create or interact with menus, as the following table shows:

Control	Description
Menu	A visual menu that can have cascading submenus
MenuBar	A horizontal bar with multiple submenus
PopUpMenuButton	A Menu control that opens when you click a button

For information on menu controls, see Chapter 11, "Using Menu-Based Controls," on page 407

Flex controls

The following table lists all the controls available with Flex:

Control	Description	For more information
Alert	Displays a pop-up alert	"Alert control" on page 351
Button	Displays a variable-size button that can include a label, an icon image, or both.	"Button control" on page 269
ButtonBar	Displays a row of related buttons with a common appearance.	"ButtonBar and ToggleButtonBar controls" on page 276
CheckBox	Shows whether a particular Boolean value is `true` (checked) or `false` (unchecked).	"CheckBox control" on page 287
ComboBox	Displays a drop-down list attached to a text field that contains a set of values.	"ComboBox control" on page 458
ColorPicker	Displays a selectable drop-down color swatch panel (palette).	"DateChooser and DateField controls" on page 296
DataGrid	Displays data in a tabular format.	"DataGrid control" on page 467

Control	Description	For more information
DateChooser	Displays a full month of days to let you select a date.	“DateChooser and DateField controls” on page 296
DateField	Displays the date with a calendar icon on its right side. When a user clicks anywhere inside the control, a DateChooser control pops up and displays a month of dates.	“DateChooser and DateField controls” on page 296
HorizontalList	Displays a horizontal list of items.	“HorizontalList control” on page 450
HRule/VRule	Displays a single horizontal rule (HRule) or vertical rule (VRule).	“HRule and VRule controls” on page 361
HSlider/VSlider	Lets users select a value by moving a slider thumb between the end points of the slider track.	“HSlider and VSlider controls” on page 310
Image	Imports GIF, JPEG, PNG, SVG, and SWF files.	“Image control” on page 325
Label	Displays a noneditable single-line field label.	“LinkButton control” on page 307
LinkBar	Displays a horizontal row of LinkButton controls that designate a series of link destinations.	“LinkBar control” on page 280
LinkButton	Displays a simple hypertext link.	“LinkButton control” on page 307
List	Displays a scrollable array of choices.	“List control” on page 440
Menu	Displays a pop-up menu of individually selectable choices, much like the File or Edit menu of most software applications.	“Handling Menu control events” on page 415
MenuBar	Displays a horizontal menu bar that contains one or more submenus of Menu controls.	“MenuBar control” on page 431
NumericStepper	Displays a dual button control that you can use to increase or decrease the value of the underlying variable.	“NumericStepper control” on page 294
ProgressBar	Provides visual feedback of how much time remains in the current operation.	“ProgressBar control” on page 356

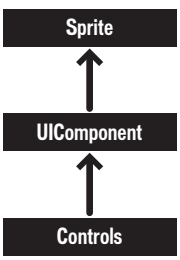
Control	Description	For more information
RadioButton	Displays a set of buttons of which exactly one is selected at any time.	"RadioButton control" on page 288
RadioButton Group	Displays a group of RadioButton controls with a single click event listener.	"Creating a group using the ‹mx:RadioButtonGroup› tag" on page 292
RichTextEditor	Includes a multiline editable text field and controls for specifying text formatting.	"RichTextEditor control" on page 398
ScrollBar (HScrollBar and VScrollBar)	Displays horizontal and vertical scroll bars.	"ScrollBar control" on page 365
SWFLoader	Displays the contents of a specified SWF file or JPEG file.	"SWFLoader control" on page 319
TabBar	Displays a horizontal row of tabs.	"TabBar control" on page 283
Text	Displays a noneditable multiline text field.	"TextInput control" on page 393
TextArea	Displays an editable text field for user input that can accept more than a single line of input.	"Using Text Controls" on page 369
TextInput	Displays an editable text field for a single line of user input. Can contain alphanumeric data, but input is interpreted as a String data type.	"TextInput control" on page 393
TileList	Displays a tiled list of items. The items are tiled in vertical columns or horizontal rows.	"TileList control" on page 454
ToggleButtonBar	Displays a row of related buttons with a common appearance.	"ButtonBar and ToggleButtonBar controls" on page 276
Tree	Displays hierarchical data arranged as an expandable tree.	"Tree control" on page 479
VideoDisplay	Incorporates streaming media into Flex applications.	"VideoDisplay control" on page 335

Working with controls

Flex controls share a common class hierarchy. Therefore, you use a similar procedure to configure all controls. This section describes the following topics:

Class hierarchy of controls

Flex controls are ActionScript objects derived from the flash.display.Sprite and mx.core.UIComponent classes, as the following example shows. Controls inherit the properties, methods, events, styles, and effects of these superclasses:



The Sprite and UIComponent classes are the base classes for all Flex components. Subclasses of the UIComponent class can have shape, draw themselves, and be invisible. Each subclass can participate in tabbing, accept low-level events like keyboard and mouse input, and be disabled so that it does not receive mouse and keyboard input.

For information on the interfaces inherited by controls from the Sprite and UIComponent classes, see Chapter 6, "Using Flex Visual Components," on page 133.

Sizing controls

This section briefly describes how Flex sizes controls. For more information on sizing components, see Chapter 8, "Sizing and Positioning Components," on page 221.

All controls define rules for determining their size in a Flex application. For example, a Button control sizes itself to fit its label text and optional icon image, while an Image control sizes itself to the size of the imported image. Each control has a default height and a default width. The default size of each standard control is specified in the description of each control.

The default size of a control is not necessarily a fixed value. For example, for a Button control, the default size is large enough to fit its label text and optional icon image. At run time, Flex calculates the default size of each control and, by default, does not resize a control from its default size.

Set the `height` or `width` attributes in MXML to percentages, such as 50%, or the `percentHeight` or `percentWidth` properties in ActionScript to percentage values, such as 50, to allow Flex to resize the control in the corresponding direction. Flex attempts to fit the control to the percentage of its parent container that you specify. If there isn't enough space available, the percentages are scaled, while retaining their relative values.

For example, you can set the width of a comments box to scale with its parent container as the parent container changes size:

```
<mx:TextInput id="comments" width="100%" height ="20"/>
```

You can also specify explicit sizes for a control. In MXML or ActionScript by setting the its `height` and `width` properties to numeric pixel values. The following example sets the height and width of the `addr2` TextInput control to 20 pixels and 100 pixels, respectively:

```
<mx:TextInput id="addr2" width="100" height ="20"/>
```

To resize a control at run time, use ActionScript to set its `width` and `height` properties. For example, the click event listener for the following Button control sets the `width` property of the `addr2` TextInput control to increase its width by 10 pixels:

```
<mx:Button id="button1" label="Slide" height="20"
  click="addr2.width+=10;"/>
```

NOTE

The preceding technique works even if the `width` property was originally set as a percentage value. The stored values of the `width` and `height` properties are always in pixels.

Many components have arbitrarily large maximum sizes, which means that Flex can make them as large as necessary to fit the requirements of your application. While some components have a defined nonzero minimum size, most have a minimum size of 0. You can use the `maxHeight`, `maxWidth`, `minHeight`, and `minWidth` properties to set explicit size ranges for each component.

Positioning controls

You place controls inside containers. Most containers have predefined layout rules that automatically determine the position of their children. The Canvas container absolutely positions its children, and the Application, and Panel containers optionally let you use absolute or container-relative positioning.

To absolutely position a control, you set its `x` and `y` properties to specific horizontal and vertical pixel coordinates within the container. These coordinates are relative to the upper-left corner of the container, where the upper-left corner is at coordinates (0,0). Values for `x` and `y` can be positive or negative integers. You can use negative values to place a control outside of the visible area of the container, and then use ActionScript to move the child to the visible area, possibly as a response to an event.

The following example places the TextInput control 150 pixels to the right and 150 pixels down from the upper-left corner of a Canvas container:

```
<mx:TextInput id="addr2" width="100" height ="20" x="150" y="150"/>
```

To reposition a control within an absolutely-positioned container at run time, you set its `x` and `y` properties. For example, the `click` event listener for the following Button control moves the TextInput control down 10 pixels from its current position:

```
<mx:Button id="button1" label="Slide" height="20" x="0" y="250"
  click="addr2.y = addr2.y+10;"/>
```

For detailed information about control positioning, including container-relative positioning, see Chapter 8, “Sizing and Positioning Components,” on page 221.

Changing the appearance of controls

Styles, skins, and fonts let you customize the appearance of controls. They describe aspects of components that you want components to have in common. Each control defines a set of styles, skins, and fonts that you can set; some are specific to a particular type of control, and others are more general.

Flex provides several different ways for you to configure the appearance of your controls. For example, you can set styles for a specific control in the control’s MXML tag, by using ActionScript, or globally for all instances of a specific control in an application by using the `<mx:Style>` tag.

A *theme* defines the look and feel of a Flex application. A theme can define something as simple as the color scheme or common font for an application, or it can be a complete reskinning of all the Flex components. The current theme for your application defines the styles that you can set on the controls within it. That means some style properties might not always be settable. For more information, see Chapter 18, “Using Styles and Themes,” on page 697.

Button control

The Button control is a commonly used rectangular button. Button controls look like they can be pressed, and have a text label, an icon, or both on their face. You can optionally specify graphic skins for each of several Button states.

You can create a normal Button control or a toggle Button control. A normal Button control stays in its pressed state for as long as the mouse button is down after you select it. A toggle Button controls stays in the pressed state until you select it a second time.

Buttons typically use event listeners to perform an action when the user selects the control. When a user clicks the mouse on a Button control, and the Button control is enabled, it dispatches a `click` event and a `buttonDown` event. A button always dispatches events such as the `mouseMove`, `mouseOver`, `mouseOut`, `rollOver`, `rollOut`, `mouseDown`, and `mouseUp` events whether enabled or disabled.

The following example shows a Button control:

You can use customized graphic skins to customize your buttons to match your application's look and functionality. You can give the Button control different image skins for the up, down, and disabled states, and the skins for these states can differ depending on whether the button is selected or not selected. The control can change the image skins dynamically.

The following example shows seven Button controls to control video recording and playback arranged in an HBox layout container. All buttons are in their up state.

The Button control has the following default properties:

Property	Default value
Default size	A size large enough to hold the label text, and any icon
Minimum size	0
Maximum size	No limit

Creating a Button control

You define a Button control in MXML by using the `<mx:Button>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. The following code creates a Button control with the label "Hello world!":

```
<?xml version="1.0"?>
<!-- controls\button\ButtonLabel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Button id="button1" label="Hello world!" width="100"/>
</mx:Application>
```

A Button control's icon, if specified, and label are centered within the bounds of the Button control. You can position the text label in relation to the icon using the `labelPlacement` property, which accepts the values `right`, `left`, `bottom`, and `top`.

Embedding an icon in a Button control

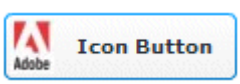
Flex lets you import graphics into your applications at both compile time and run time. Button icons must be embedded at compile time rather than referenced at run time. You can use the `@Embed` syntax in the `icon` property value to embed any GIF, JPEG, PNG, SVG, or SWF file, or you can bind to an image that you defined within a script block by using `[Embed]` metadata. If you must reference your button graphic at run time, you can use an `<mx:Image>` tag instead of an `<mx:Button>` tag.

For more information on embedding resources, see Chapter 30, "Embedding Assets," on page 1113.

The following code example creates a Button control with a label and an icon:

```
<?xml version="1.0"?>
<!-- controls\button\ButtonLabelIcon.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Button
        label="Icon Button"
        icon="@Embed(source='assets/logo.jpg')"/>
</mx:Application>
```

The icon is in the assets subdirectory of the directory containing the application file. This results in a button with the icon displayed to the left of the label text:



For an overview of resource embedding, see Chapter 30, "Embedding Assets," on page 1113.

Sizing a Button control

By default, Flex stretches the Button control width to fit the size of its label, any icon, plus 6 pixels of padding around the icon. You can override this default width by explicitly setting the `width` property of the Button control to a specific value or to a percentage of its parent container. If you specify a percentage value, the button resizes between its minimum and maximum widths as the size of its parent container changes.

If you explicitly size a Button control so that it is not large enough to accommodate its label, the label is truncated and terminated by an ellipses (...). The full label displays as a tooltip when you move the mouse over the Button control. If you have also set a tooltip using the `tooltip` property, the tooltip is displayed rather than the label text.

Text that is vertically larger than the Button control is also clipped. If you explicitly size a Button control so that it is not large enough to accommodate its icon, icons larger than the Button control extend outside the Button control's bounding box.

Button control user interaction

When a user clicks the mouse on a Button control, the Button control dispatches a `click` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\button\ButtonClick.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Input field. -->
    <mx:TextInput id="myInput" width="150" text=""/>

    <!-- Button control that triggers the copy. -->
    <mx:Button id="myButton" label="Copy Text"
        click="myText.text=myInput.text;"/>

    <!-- Output text box. -->
    <mx:TextArea id="myText" text="" width="150" height="20"/>
</mx:Application>
```

In this example, clicking the Button control copies the text from the TextInput control to the TextArea control.

If a Button control is enabled, it behaves as follows:

- When the user moves the mouse pointer over the Button control, the Button control displays its rollover appearance.
- When the user clicks the Button control, focus moves to the control and the Button control displays its pressed appearance. When the user releases the mouse button, the Button control returns to its rollover appearance.

- If the user moves the mouse pointer off the Button control while pressing the mouse button, the control's appearance returns to the original state and it retains focus.
- If the `toggle` property is set to `true`, the state of the Button control does not change until the user releases the mouse button over the control.

If a Button control is disabled, it displays its disabled appearance, regardless of user interaction. In the disabled state, all mouse or keyboard interaction is ignored.

Skinning a Button control

You can specify a set of up to eight different image skin properties, where each property corresponds to a different button state. These skins determine the basic appearance of the buttons. You can specify images for each of the following button states:

- Up (the mouse is not over the control)
- Down (the mouse is over the control and the mouse button is pressed)
- Over (the mouse hovers over the control)
- Disabled
- Selected and up
- Selected and down
- Selected and over
- Selected and disabled

Specifying image skins

You specify the default appearance of a Button control by specifying an up state image (the `upSkin` property). All other states use the up state image or the image from another state as their default. For example, if you do not specify an image for the down state, Flex uses the image specified for the over state; if you don't specify an image for the over state, Flex uses the image for the up state. The selected states are used only for toggle buttons that are selected (pressed).

The skin image determines the appearance of the button, including its shape. The image can be GIF, JPEG, PNG, SVG, or SWF file. You can create the skins as independent image files, or incorporate multiple images in a single SWF file.

Flex must embed the button images in the application's SWF file at compile time; you cannot download images from the server at run time. To embed the image, use the `@Embed` MXML compiler directive. The following code example shows how to use a GIF file as the up (default) button image:

```
upSkin="@Embed(source='assets/buttonUp.gif')"
```

The following code example creates a toggle button with an image for up, down, over, and disabled states. The button has both a label and an icon.

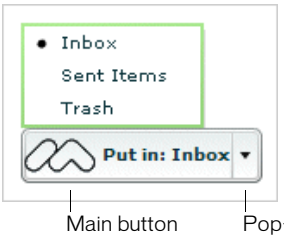
```
<?xml version="1.0"?>
<!-- controls\button\ButtonSkin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

<mx:Button label="Image Button"
    toggle="true"
    color="0xFFFFAA"
    textRollOverColor="0xAAAA55"
    textSelectedColor="0xFFFF00"
    upSkin="@Embed(source='assets/buttonUp.gif')"
    overSkin="@Embed(source='assets/buttonOver.gif')"
    downSkin="@Embed(source='assets/buttonDown.gif')"
    disabledSkin="@Embed(source='assets/buttonDisabled.gif')"
    icon="@Embed(source='assets/logo.gif')"/>
</mx:Application>
```

PopUpButton control

The PopUpButton control consists of two horizontal buttons: a main button, and a smaller button called the pop-up button, which only has an icon. The main button is a Button control.

The pop-up button, when clicked, opens a second control called the pop-up control. Clicking anywhere outside the PopUpButton control, or in the pop-up control, closes the pop-up control

The PopUpButton control adds a flexible pop-up control interface to a Button control. One common use for the PopUpButton control is to have the pop-up button open a List control or a Menu control that changes the function and label of the main button, as the following example shows using a Menu control:



In this example, the user can choose whether the button puts mail in the Inbox, the Sent Items folder, or the Trash folder, by selecting from the pop-up menu that appears when the user clicks the small pop-up button to the right of the main button. The text on the main button indicates the action it performs, and the text changes each time the user selects a different item from the menu.

The PopUpButton control is not limited to displaying menus; it can display any Flex control as the pop-up control. A workflow application that lets users send a document for review, for example, could use a Tree control as a visual indication of departmental structure. The PopUpButton control's pop-up button would display the tree, from which the user could pick the message recipients.

The control that pops up does not have to affect the main button's appearance or action; it can have an independent action, instead. You could create an undo PopUpButton control, for example, where the main button undoes only the last action, and the pop-up control is a List control that lets users undo multiple actions by selecting them.

The PopUpButton control is a subclass of the Button control and inherits all of its properties, styles, events, and methods, with the exception of the `toggle` property and the styles used for a selected button.

The control has the following characteristics:

- The `popUp` property specifies the pop-up control (for example, List or Menu).
- The `open()` and `close()` methods lets you open and close the pop-up control programmatically, rather than using the pop-up button.
- The `open` and `close` events are dispatched when the pop-up control opens and closes.
- You use the `popUpSkin` and `arrowButtonWidth` style properties to define the PopUpButton control's appearance.

For detailed descriptions, see PopUpButton in *Adobe Flex 2 Language Reference*.

The PopUpButton control has the following default properties:

Property	Default value
Default size	Sufficient to accommodate the label and icon on the main button and the icon on the pop-up button
Minimum size	0
Maximum size	Undefined

Creating a PopUpButton control

You use the `<mx:PopUpButton>` tag to define a PopUpButton control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

In the following example, you use the PopUpButton control to open a Menu control. Once opened, the Menu control, or any pop-up control, functions just as it would normally. You define an event listener for the Menu control's change event to recognize when the user selects a menu item, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\button\PopUpButtonMenu.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="600" width="600">

    <mx:Script>
        <![CDATA[
            import mx.controls.*;
            import mx.events.*;

            private var myMenu:Menu;

            // Initialize the Menu control,
            // and specify it as the pop up object
            // of the PopUpButton control.
            private function initMenu():void {
                myMenu = new Menu();
                var dp:Object = [
                  {label: "New Folder"},
                  {label: "Sent Items"},
                  {label: "Inbox"}
                ];
                myMenu.dataProvider = dp;
                myMenu.addEventListener("itemClick", changeHandler);
                popB.popUp = myMenu;
            }

           // Define the event listener for the Menu control's change event.
            private function changeHandler(event:MenuEvent):void {
                var label:String = event.label;
                popTypeB.text=String("Moved to " + label);
                popB.label = "Put in: " + label;
                popB.close();
            }
        ]]>
    </mx:Script>

    <mx:VBox>
```

```
        <mx:Label text="Main button mimics the last selected menuItem."/>
        <mx:PopUpButton id="popB"
            label="Edit"
            width="135"
            creationComplete="initMenu();"/>

        <mx:Spacer height="50"/>
        <mx:TextInput id="popTypeB"/>
    </mx:VBox>
</mx:Application>
```

User interaction

You navigate the PopUpButton control using the mouse as follows:

- Moving the mouse over any part of the PopUpButton control highlights the button border and the main button or the pop-up button.
- Clicking the button dispatches the `click` event.
- Clicking the pop-up button pops up the pop-up control and dispatches an `open` event.
- Clicking anywhere outside the PopUpButton control, or in the pop-up control, closes the pop-up control and dispatches a `close` event.

The following keystrokes let users navigate the PopUpButton control:

Key	Use
Spacebar	Behaves like clicking the main button.
Control+Down Arrow	Opens the pop-up control and initiates an `open` event. The pop-up control's keyboard handling takes effect.
Control+Up Arrow	Closes the pop-up control and initiates a `close` event.

NOTE

You cannot use the Tab key to leave an opened pop-up control; you must make a selection or close the control with the Control+Up Arrow key combination.

ButtonBar and ToggleButtonBar controls

The ButtonBar and ToggleButtonBar controls define a horizontal or vertical row of related buttons with a common appearance. The controls define a single event, the `itemClick` event, that is dispatched when any button in the control is selected.

The ButtonBar control defines group of buttons that do not retain a selected state. When you select a button in a ButtonBar control, the button changes its appearance to the selected state; when you release the button, it returns to the deselected state.

The ToggleButtonBar control defines a group buttons that maintain their state, either selected or deselected. Only one button in the ToggleButtonBar control can be in the selected state. That means when you select a button in a ToggleButtonBar control, the button stays in the selected state until you select a different button.

If you set the `unselectable` property of the ToggleButtonBar control to `true`, selecting the currently selected button deselects it. By default the `unselectable` properties `false`.

The following image shows an example of a ButtonBar control that defines a set of buttons:

The following image shows an example of a ToggleButtonBar control that defines a set of buttons, where the Dreamweaver button is the currently selected button in the control:

A ButtonBar and ToggleButtonBar control have the following default properties:

Property	Default value
Preferred size	Wide enough to contain all buttons with their label text and icons, if any, plus any padding and separators, and high enough to accommodate the button height.
Control resizing rules	The controls do not resize by default. Specify percentage sizes if you want your ButtonBar to resize based on the size of its parent container.
Padding	0 pixels for the `top`, `bottom`, `left`, and `right` properties.

Creating a ButtonBar control

You create a ButtonBar control in MXML using the `<mx:ButtonBar>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\BBarSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:ButtonBar borderStyle="solid" horizontalGap="5">
        <mx:dataProvider>
            <mx:String>Flash</mx:String>
            <mx:String>Director</mx:String>
            <mx:String>Dreamweaver</mx:String>
            <mx:String>ColdFusion</mx:String>
        </mx:dataProvider>
    </mx:ButtonBar>
</mx:Application>
```

This example creates a row of four Button controls, as shown in the image in the section "ButtonBar and ToggleButtonBar controls" on page 276.

To create a ToggleButtonBar control, replace the `<mx:ButtonBar>` tag with the `<mx:ToggleButtonBar>` tag. Otherwise, the syntax is the same for both controls.

You use the `dataProvider` property to specify the labels of the four buttons. You can also populate the `dataProvider` property with an Array of Objects; where each object can have up to three fields: `label`, `icon`, and `tooltip`.

In the following example, you use an Array of Objects to specify a label and icon for each button:

```
<?xml version="1.0"?>
<!-- controls\bar\BBarLogo.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:ButtonBar borderStyle="solid" horizontalGap="5">
        <mx:dataProvider>
            <mx:Object label="Flash"
                icon="@Embed(source='assets/Flashlogo.gif')"/>
            <mx:Object label="Director"
                icon="@Embed(source='assets/Dirlogo.gif')"/>
            <mx:Object label="Dreamweaver"
                icon="@Embed(source='assets/Dlogo.gif')"/>
            <mx:Object label="ColdFusion"
                icon="@Embed(source='assets/CFlogo.gif')"/>
        </mx:dataProvider>
    </mx:ButtonBar>
</mx:Application>
```

A ButtonBar or ToggleButtonBar control creates Button controls based on the value of its `dataProvider` property. Even though ButtonBar and ToggleButtonBar are subclasses of Container, do not use methods such as `Container.addChild()` and `Container.removeChild()` to add or remove Button controls. Instead, use methods such as `addItem()` and `removeItem()` to manipulate the `dataProvider` property. A ButtonBar or ToggleButtonBar control automatically adds or removes the necessary children based on changes to the `dataProvider` property.

Handling ButtonBar events

The ButtonBar and ToggleButtonBar controls dispatch a `itemClick` event when you select a button. The event object passed to the event listener is of type ItemClickEvent. From within the event listener, you can access properties of the event object to determine the index of the selected button, where the first button is at index 0, and other information. For more information on the event object, see the description of the ItemClickEvent class in *Adobe Flex 2 Language Reference*.

The ButtonBar control in the following example defines an event listener for the `itemClick` event:

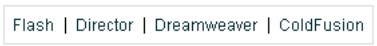
```
<?xml version="1.0"?>
<!-- controls\bar\BBarEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
            import mx.events.ItemClickEvent;

            private function clickHandler(event:ItemClickEvent):void {
                myTA.text="Selected button index: " +
                    String(event.index) + "\n" +
                    "Selected button label: " +
                    event.label;
            }
        ]]>
    </mx:Script>

    <mx:TextArea id="myTA" width="200" height="100"/>

    <mx:ButtonBar
        borderStyle="solid"
        horizontalGap="5"
        itemClick="clickHandler(event);">
        <mx:dataProvider>
            <mx:String>Flash</mx:String>
            <mx:String>Director</mx:String>
            <mx:String>Dreamweaver</mx:String>
            <mx:String>ColdFusion</mx:String>
        </mx:dataProvider>
    </mx:ButtonBar>
</mx:Application>
```

In this example, the event listener displays the index and label of the selected button in a TextArea control in response to a `itemClick` event.

LinkBar control

A LinkBar control defines a horizontal or vertical row of LinkButton controls that designate a series of link destinations. You typically use a LinkBar control to control the active child container of a ViewStack container, or to create a standalone set of links.

The following shows an example of a LinkBar control that defines a set of links:



A LinkBar control has the following default properties:

Property	Default value
Preferred size	A width wide enough to contain all label text, plus any padding and separators, and the height of the tallest child.
Control resizing rules	LinkBar controls do not resize by default. Specify percentage sizes if you want your LinkBar to resize based on the size of its parent container.
Padding	2 pixels for the `top`, `bottom`, `left`, and `right` properties.

Creating a LinkBar control

One of the most common uses of a LinkBar control is to control the active child of a ViewStack container. For an example, see "ViewStack navigator container" on page 628.

You can also use a LinkBar control on its own to create a set of links in your application. In the following example, you define a `itemClick` handler for the LinkBar control to respond to user input, and use the `dataProvider` property of the LinkBar to specify its label text. Use the following example code to create the LinkBar control shown in the previous image:

```
<?xml version="1.0"?>
<!-- controls\bar\LBarSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:LinkBar borderStyle="solid"
        itemClick="navigateToURL(new URLRequest('http://www.adobe.com/' +
            String(event.label).toLowerCase()), '_blank');">
        <mx:dataProvider>
            <mx:String>Flash</mx:String>
            <mx:String>Director</mx:String>
            <mx:String>Dreamweaver</mx:String>
            <mx:String>ColdFusion</mx:String>
        </mx:dataProvider>
    </mx:LinkBar>
</mx:Application>
```

In this example, you use the `<mx:dataProvider>` and `<mx:Array>` tags to define the label text. The event object passed to the `itemClick` handler contains the label selected by the user. The handler for the `itemClick` event constructs an HTTP request to the Adobe website based on the label, and opens that page in a new browser window.

You can also bind data to the <mx:dataProvider> tag to populate the LinkBar control, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\LBarBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var linkData:ArrayCollection = new ArrayCollection([
                "Flash", "Director", "Dreamweaver", "ColdFusion"
            ]);
        ]]>
    </mx:Script>

    <mx:LinkBar
        horizontalAlign="right"
        borderStyle="solid"
        itemClick="navigateToURL(new URLRequest('http://www.adobe.com/' +
            String(event.label).toLowerCase()), '_blank');">
        <mx:dataProvider>
            {linkData}
        </mx:dataProvider>
    </mx:LinkBar>
</mx:Application>
```

In this example, you define the data for the LinkBar control as a variable in ActionScript, and then you bind that variable to the <mx:dataProvider> tag. You could also bind to the <mx:dataProvider> tag from a Flex data model, from a web service response, or from any other type of data model.

A LinkBar control creates LinkButton controls based on the value of its dataProvider property. Even though LinkBar is a subclass of Container, do not use methods such as Container.addChild() and Container.removeChild() to add or remove LinkButton controls. Instead, use methods such as addItem() and removeItem() to manipulate the dataProvider property. A LinkBar control automatically adds or removes the necessary children based on changes to the dataProvider property.

TabBar control

A TabBar control defines a horizontal or vertical row of tabs. The following shows an example of a TabBar control:

As with the LinkBar control, you can use a TabBar control to control the active child container of a ViewStack container. The syntax for using a TabBar control to control the active child of a ViewStack container is the same as for a LinkBar control. For an example, see "ViewStack navigator container" on page 628.

While a TabBar control is similar to a TabNavigator container, it does not have any children. For example, you use the tabs of a TabNavigator container to select its visible child container. You can use a TabBar control to set the visible contents of a single container to make that container's children visible or invisible based on the selected tab.

A TabBar control has the following default properties:

Property	Default value
Preferred size	A width wide enough to contain all label text, plus any padding, and a height tall enough for the label text. The default tab height is determined by the font, style, and skin applied to the control. If you set an explicit height using the `tabHeight` property, that value overrides the default value.
Control resizing rules	TabBar controls do not resize by default. Specify percentage sizes if you want your TabBar to resize based on the size of its parent container.
Padding	0 pixels for the `left` and `right` properties.

Creating a TabBar control

You use the `<mx:TabBar>` tag to define a TabBar control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the data for the TabBar control using the <mx:dataProvider> and <mx:Array> child tags of the <mx:TabBar> tag. The <mx:dataProvider> tag lets you specify data in several different ways. In the simplest case for creating a TabBar control, you use the <mx:dataProvider>, <mx:Array>, and <mx:String> tags to specify the text for each tab, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TabBar>
        <mx:dataProvider>
            <mx:String>Alabama</mx:String>
            <mx:String>Alaska</mx:String>
            <mx:String>Arkansas</mx:String>
        </mx:dataProvider>
    </mx:TabBar>
</mx:Application>
```

The <mx:String> tags define the text for each tab in the TabBar control.

You can also use the <mx:Object> tag to define the entries as an array of objects, where each object contains a label property and an associated data value, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarObject.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TabBar>
        <mx:dataProvider>
            <mx:Object label="Alabama" data="Montgomery"/>
            <mx:Object label="Alaska" data="Juneau"/>
          <mx:Object label="Arkansas" data="Little Rock"/>
        </mx:dataProvider>
    </mx:TabBar>
</mx:Application>
```

The label property contains the state name and the data property contains the name of its capital. The data property lets you associate a data value with the text label. For example, the label text could be the name of a color, and the associated data value could be the numeric representation of that color.

By default, Flex uses the value of the `label` property to define the tab text. If the object does not contain a `label` property, you can use the `labelField` property of the TabBar control to specify the property name containing the tab text, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarLabel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TabBar labelField="state">
        <mx:dataProvider>
            <mx:Object state="Alabama" data="Montgomery"/>
            <mx:Object state="Alaska" data="Juneau"/>
          <mx:Object state="Arkansas" data="Little Rock"/>
        </mx:dataProvider>
    </mx:TabBar>
</mx:Application>
```

Passing data to a TabBar control

Flex lets you populate a TabBar control from an ActionScript variable definition or from a Flex data model. When you use a variable, you can define it to contain one of the following:

- A label (string)
- A label (string) paired with data (scalar value or object)

The following example populates a TabBar control from a variable:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarVar.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var STATE_ARRAY:ArrayCollection = new ArrayCollection([
                {label:"Alabama", data:"Montgomery"},
                {label:"Alaska", data:"Juneau"},
                {label:"Arkansas", data:"LittleRock"}
            ]);
        ]]>
    </mx:Script>

    <mx:TabBar >
        <mx:dataProvider>
            {STATE_ARRAY}
        </mx:dataProvider>
    </mx:TabBar>
</mx:Application>
```

You can also bind a Flex data model to the `dataProvider` property. For more information on using data models, see Chapter 39, "Storing Data," on page 1269.

Handling TabBar control events

The TabBar control defines a `itemClick` event that is broadcast when a user selects a tab. The event object contains the following properties:

- `label` String containing the label of the selected tab.
- `index` Number containing the index of the selected tab. Indexes are numbered from 0 to *n* - 1, where *n* is the total number of tabs. The default value is 0, corresponding to the first tab.

The following example code shows a handler for the `itemClick` event for this TabBar control:

```
<?xml version="1.0"?>
<!-- controls\bar\TBarEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.events.ItemClickEvent;
            import mx.controls.TabBar;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var STATE_ARRAY:ArrayCollection = new ArrayCollection([
                {label:"Alabama", data:"Montgomery"},
                {label:"Alaska", data:"Juneau"},
                {label:"Arkansas", data:"LittleRock"}
            ]);

            private function clickEvt(event:ItemClickEvent):void {
                // Access target TabBar control.
                var targetComp:TabBar = TabBar(event.currentTarget);
                forClick.text="label is: " + event.label + " index is: " +
                    event.index + " capital is: " +
                    targetComp.dataProvider[event.index].data;
            }
        ]]>
    </mx:Script>

    <mx:TabBar id="myTB" itemClick="clickEvt(event);">
        <mx:dataProvider>
            {STATE_ARRAY}
        </mx:dataProvider>
```

```
    </mx:TabBar>

    <mx:TextArea id="forClick" width="150"/>
</mx:Application>
```

In this example, every `itemClick` event updates the TextArea control with the tab label, selected index, and the selected data from the TabBar control's `dataProvider` Array.

CheckBox control

The CheckBox control is a commonly used graphical control that can contain a check mark or be unchecked (empty). You can use CheckBox controls wherever you need to gather a set of `true` or `false` values that aren't mutually exclusive.

You can add a text label to a CheckBox control and place it to the left, right, top, or bottom of the check box. Flex clips the label of a CheckBox control to fit the boundaries of the control.

The following image shows a checked CheckBox control:

For the code used to generate this example, see "Creating a CheckBox control" on page 288.

When a user clicks a CheckBox control or its associated text, the CheckBox control changes its state from checked to unchecked, or from unchecked to checked.

A CheckBox control can have one of two disabled states, checked or unchecked. By default, a disabled CheckBox control displays a different background and check mark color than an enabled CheckBox control.

The CheckBox control has the following default properties:

Property	Default value
Default size	A size large enough to hold the label
Minimum size	0
Maximum size	No limit

Creating a CheckBox control

You use the `<mx:CheckBox>` tag to define a CheckBox control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\checkbox\CBSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:VBox>
        <mx:CheckBox width="100" label="Employee?"/>
    </mx:VBox>
</mx:Application>
```

You can also use the `selected` property to generate a checkbox that is checked by default:

```
<?xml version="1.0"?>
<!-- controls\checkbox\CBSelected.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:VBox>
        <mx:CheckBox width="100" label="Employee?" selected="true"/>
    </mx:VBox>
</mx:Application>
```

CheckBox control user interaction

When a CheckBox control is enabled and the user clicks it, the control receives focus and displays its checked or unchecked appearance, depending on its initial state. The entire area of the CheckBox control is the click area; if the CheckBox control's text is larger than its icon, the clickable regions are above and below the icon.

If the user moves the mouse pointer outside the area of the CheckBox control or its label while pressing the mouse button, the appearance of the CheckBox control returns to its original state and the control retains focus. The state of the CheckBox control does not change until the user releases the mouse button over the control.

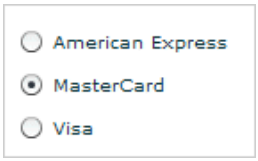
Users cannot interact with a CheckBox control when it is disabled.

RadioButton control

The RadioButton control is a single choice in a set of mutually exclusive choices. A RadioButton group is composed of two or more RadioButton controls with the same group name. Only one member of the group can be selected at any given time. Selecting an unselected group member deselects the currently selected RadioButton control in the group.

About the RadioButton control

The following example shows a RadioButton group with three RadioButton controls:



For the code used to generate this example, see "Creating a RadioButton control" on page 289.

The RadioButton control has the following default properties:

Property	Default value
Default size	Wide enough to display the text label of the control
Minimum size	0
Maximum size	Undefined

Creating a RadioButton control

You define a RadioButton control in MXML using the `<mx:RadioButton>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\button\RBSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:RadioButton groupName="cardtype"
        id="americanExpress"
        label="American Express"
        width="150"/>
    <mx:RadioButton groupName="cardtype"
        id="masterCard"
        label="MasterCard"
        width="150"/>
    <mx:RadioButton groupName="cardtype"
        id="visa"
        label="Visa"
        width="150"/>
</mx:Application>
```

This code results in the application shown in "About the RadioButton control" on page 289.

For each RadioButton control in the group, you can optionally define an event listener for the button's `click` event. When a user selects a RadioButton control, Flex calls the event listener associated with the button for the `click` event, as the following code example shows:

```
<?xml version="1.0"?>
<!-- controls\button\RBEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private function handleAmEx(event:Event):void {
                // Handle event.
                myTA.text="Got Amex";
            }

            private function handleMC(event:Event):void {
                // Handle event.
                myTA.text="Got Amex";
            }

            private function handleVisa(event:Event):void {
                // Handle event.
                myTA.text="Got Amex";
            }
        ]]>
    </mx:Script>

    <mx:RadioButton groupName="cardtype"
        id="americanExpress"
        label="American Express"
        width="150"
        click="handleAmEx(event);"/>
    <mx:RadioButton groupName="cardtype"
        id="masterCard"
        label="MasterCard"
        width="150"
        click="handleMC(event);"/>
    <mx:RadioButton groupName="cardtype"
        id="visa"
        label="Visa"
        width="150"
        click="handleVisa(event);"/>

    <mx:TextArea id="myTA"/>
</mx:Application>
```

RadioButton user interaction

If a RadioButton control is enabled, when the user moves the mouse pointer over an unselected RadioButton control, the button displays its rollover appearance. When the user clicks an unselected RadioButton control, the input focus moves to the control and the button displays its false pressed appearance. When the mouse button is released, the button displays the true state appearance. The previously selected RadioButton control in the group returns to its false state appearance.

If the user moves the mouse pointer off the RadioButton control while pressing the mouse button, the control's appearance returns to the false state and the control retains input focus.

If a RadioButton control is not enabled, the RadioButton control and RadioButton group display the disabled appearance, regardless of user interaction. In the disabled state, all mouse or keyboard interaction is ignored.

The RadioButton and RadioButtonGroup controls have the following keyboard navigation features:

Key	Action
Control+Arrow keys	Move focus among the buttons without selecting a button.
Spacebar	Select a button.

Creating a group using the <mx:RadioButtonGroup> tag

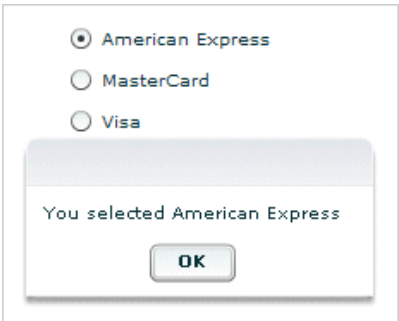
The previous example created a RadioButton group using the `groupName` property of each RadioButton control. You can also create a RadioButton group using the RadioButtonGroup control, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\button\RBGroupSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.events.ItemClickEvent;

            private function handleCard(event:ItemClickEvent):void {
                //Handle event.
            }
        ]]>
    </mx:Script>

    <mx:RadioButtonGroup id="cardtype" itemClick="handleCard(event);"/>
    <mx:RadioButton groupName="cardtype"
        id="americanExpress"
        value="AmEx"
        label="American Express"
        width="150"/>
    <mx:RadioButton groupName="cardtype"
        id="masterCard"
        value="MC"
        label="MasterCard"
        width="150"/>
    <mx:RadioButton groupName="cardtype"
        id="visa"
        value="Visa"
        label="Visa"
        width="150"/>
</mx:Application>
```

In this example, you use the `id` property of the `<mx:RadioButtonGroup>` tag to define the group name and the single `itemClick` event listener for all buttons in the group. The `id` property is required when you use the `<mx:RadioButtonGroup>` tag. The `itemClick` event listener for the group can determine which button was selected, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\button\RBGroupEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.controls.Alert;
            import mx.events.ItemClickEvent;

            private function handleCard(event:ItemClickEvent):void {
                if (event.currentTarget.selectedValue == "AmEx") {
                    Alert.show("You selected American Express")
                }
                else {
                    if (event.currentTarget.selectedValue == "MC") {
                        Alert.show("You selected Master Card")
                    }
                    else {
                        Alert.show("You selected Visa")
                    }
                }
            }
        ]]>
    </mx:Script>

    <mx:RadioButtonGroup id="cardtype" itemClick="handleCard(event);"/>
    <mx:RadioButton groupName="cardtype"
        id="americanExpress"
        value="AmEx"
        label="American Express"
        width="150"/>
    <mx:RadioButton groupName="cardtype"
        id="masterCard"
        value="MC"
        label="MasterCard"
        width="150"/>
    <mx:RadioButton groupName="cardtype"
        id="visa"
        value="Visa"
        label="Visa"
        width="150"/>
</mx:Application>
```

This code results in the following output when you select the American Express button:



In the `itemClick` event listener, the `selectedValue` property of the RadioButtonGroup control in the event object is set to the value of the `value` property of the selected RadioButton control. If you omit the `value` property, Flex sets the `selectedValue` property to the value of the `label` property.

You can still define a `click` event listener for the individual buttons, even though you also define one for the group.

NumericStepper control

You can use the NumericStepper control to select a number from an ordered set. The NumericStepper control consists of a single-line input text field and a pair of arrow buttons for stepping through the valid values; you can also use the Up Arrow and Down Arrow keys to cycle through the values.

The following example shows a NumericStepper control:

For the code used to create this image, see "Creating a NumericStepper control" on page 295.

If the user clicks the up arrow, the value displayed is increased by one unit of change. If the user holds down the arrow, the value increases or decreases until the user releases the mouse button. When the user clicks the arrow, it is highlighted to provide feedback to the user.

Users can also type a legal value directly into the text field. Although editable ComboBox controls provide similar functionality, NumericStepper controls are sometimes preferred because they do not require a drop-down list that can obscure important data.

NumericStepper control arrows always appear to the right of the text field.

The NumericStepper control has the following default properties:

Property	Default value
Default size	Wide enough to display the maximum number of digits used by the control
Minimum size	Based on the size of the text
Maximum size	Undefined

Creating a NumericStepper control

You define a NumericStepper control in MXML using the `<mx:NumericStepper>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\numericstepper\NumStepSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:NumericStepper id="nstepper1" value="6" stepSize="2"/>
</mx:Application>
```

Sizing a NumericStepper control

The up and down arrow buttons in the NumericStepper control do not change size when the control is resized. If the NumericStepper control is sized greater than the default height, the associated stepper buttons appear pinned to the top and the bottom of the control.

User interaction

If the user clicks the up or down arrow button, the value displayed is increased by one unit of change. If the user presses either of the arrow buttons for more than 200 milliseconds, the value in the input field increases or decreases, based on step size, until the user releases the mouse button or the maximum or minimum value is reached.

Keyboard navigation

The NumericStepper control has the following keyboard navigation features:

Key	Description
Down Arrow	Value decreases by one unit.
Up Arrow	Value increases by one unit.

Key	Description
Left Arrow	Moves the insertion point to the left within the NumericStepper control's text field.
Right Arrow	Moves the insertion point to the right within the Numeric Stepper control's text field.

In order to use the keyboard to navigate through the stepper, it must have focus.

DateChooser and DateField controls

The DateChooser and DateField controls let users select dates from graphical calendars. The DateChooser control user interface is the calendar. The DateField control has a text field that uses a date chooser popup to select the date; as a result. The DateField properties are a superset of the DateChooser properties.

For complete reference information, see DateChooser and DateField in *Adobe Flex 2 Language Reference*.

About the DateChooser control

The DateChooser control displays the name of a month, the year, and a grid of the days of the month, with columns labeled for the days of the week. This control is useful in applications where you want a continually visible calendar. The user can select a single date from the grid. The control contains forward and back arrow buttons to let you change the month and year. You can disable the selection of certain dates, and limit the display to a range of dates.

The following image shows a DateChooser control:

Changing the displayed month does not change the selected date. Therefore, the currently selected date might not always be visible. The DateChooser control resizes as necessary to accommodate the width of the weekday headings. Therefore, if you use day names, instead of letters, as headings, the calendar will be wide enough to show the full day names.

The DateChooser control has the following default properties:

Property	Default value
Default size	A size large enough to hold the calendar, and wide enough to display the day names
Minimum size	0
Maximum size	No limit

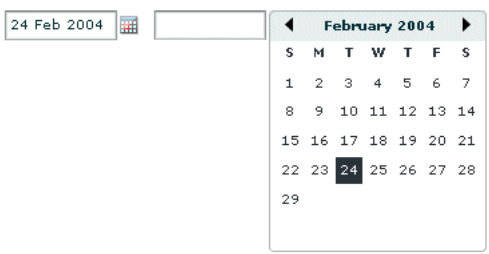
About the DateField control

The DateField control is a text field that displays the date with a calendar icon on its right side. When a user clicks anywhere inside the bounding box of the control, a date chooser that is identical to the DateChooser control pops up. If no date has been selected, the text field is blank and the current month is displayed in the date chooser.

When the date chooser is open, users can click the month scroll buttons to scroll through months and years, and select a date. When the user selects a date, the date chooser closes and the text field displays the selected date.

This control is useful in applications where you want a calendar selection tool, but want to minimize the space the date information takes up.

The following example shows two images of a DateField control. On the left is a control with the date chooser closed; the calendar icon appears on the right side of the text box. To the right is a DateField control with the date chooser open.



You can use the DateField control anywhere you want a user to select a date. For example, you can use a DateField control in a hotel reservation system, with certain dates selectable and others disabled. You can also use the DateField control in an application that displays current events, such as performances or meetings, when a user selects a date.

The DateField has the same default properties as the DateChooser for its expanded date chooser. It has the following default properties for the collapsed control:

Property	Default value
Default size	A size large enough to hold the formatted date and the calendar icon
Minimum size	0
Maximum size	No limit

Creating a DateChooser or DateField control

You define a DateChooser control in MXML using the `<mx:DateChooser>` tag. You define a DateField control in MXML using the `<mx:DateField>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

The following example creates a DateChooser control; to create a DateField control, simply change `<mx:DateChooser>` to `<mx:DateField>`. The example uses the `change` event of the DateChooser control to display the selected date in several different formats.

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

        import mx.events.CalendarLayoutChangeEvent;

        private function useDate(eventObj:CalendarLayoutChangeEvent):void {
            // Make sure selectedDate is not null.
            if (eventObj.currentTarget.selectedDate == null) {
                return
            }

            //Access the Date object from the event object.
            day.text=eventObj.currentTarget.selectedDate.getDay();
            date.text=eventObj.currentTarget.selectedDate.getDate();
            month.text=eventObj.currentTarget.selectedDate.getMonth();
            year.text=eventObj.currentTarget.selectedDate.getFullYear();
            wholeDate.text=
                eventObj.currentTarget.selectedDate.getFullYear() +
                "/" +
                (eventObj.currentTarget.selectedDate.getMonth()+1) +
                "/" + eventObj.currentTarget.selectedDate.getDate();
        }
        ]]>
    </mx:Script>
```

```
    <mx:DateChooser id="date1" change="useDate(event)"/>

    <mx:Form>
        <mx:FormItem label="Day">
            <mx:TextInput id="day" width="100"/>
        </mx:FormItem>
        <mx:FormItem label="Day of month">
            <mx:TextInput id="date" width="100"/>
        </mx:FormItem>
        <mx:FormItem label="Month">
            <mx:TextInput id="month" width="100"/>
        </mx:FormItem>
        <mx:FormItem label="Year">
            <mx:TextInput id="year" width="100"/>
        </mx:FormItem>
        <mx:FormItem label="Date">
            <mx:TextInput id="wholeDate" width="300"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Notice that the first line of the event listener determines if the `selectedDate` property is null. This check is necessary because selecting the currently selected date deselects it, sets the `selectedDate` property to null, then dispatches the `change` event.

NOTE The code that determines the value of the wholeDate field adds 1 to the month number because the DateChooser control uses a zero-based month system, where January is month 0 and December is month 11.

Using the Date class

The DateChooser and DateField controls use the `selectedDate` property to store the currently selected date, as an object of type Date. You can create Date objects to represent date and time values, or access the Date in the `selectedDate` property.

The Date class has many methods that you can use to manipulate a date. For more information on the Date class, see the *Adobe Flex 2 Language Reference*.

In MXML you can create and configure a Date object using the `<mx:Date>` tag. This tag exposes the setter methods of the Date class as MXML properties so that you can initialize a Date object. For example, the following code creates a DateChooser control, and sets the selected date to April 10, 2005 (notice that months are indexed starting at 0 for the DateChooser control):

```
<mx:DateChooser id="date1">
     <mx:selectedDate>
       <mx:Date month="9" date="10" year="2005"/>
```

```
    </mx:selectedDate>
</mx:DateChooser>
```

The following example uses inline ActionScript to set the initial selected date for a DateField control:

```
<mx:DateField id="date3" selectedDate="{new Date (2005, 9, 10)}"/>
```

You can also set the `selectedDate` property in a function, as the following example shows:

```
<mx:Script>
  <![CDATA[
    private function initDC():void {
      date2.selectedDate=new Date (2003, 3, 10);
    }
  ]]>
</mx:Script>

<mx:DateChooser id="date2" creationComplete="initDC();"/>
```

You can use property notation to access the ActionScript setter and getter methods of the `selectedDate` property Date object. For example, the following line displays the four-digit year of the selected date in a text box.

```
<mx:TextInput text="{date1.selectedDate.fullYear}"/>
```

Specifying header, weekday, and today's day text styles

The following date chooser properties let you specify text styles for regions of the control:

- `headerStyleName`
- `weekDayStyleName`
- `todayStyleName`

These properties let you specify styles for the text in the header, week day list and today's date. You cannot use these properties to set non-text styles such as `todayColor`.

The following example defines a DateChooser control that has bold, blue header text in a 16-pixel Times New Roman font. The day of week headers are in bold, italic, green, 15-pixel Courier text, and today's date is bold, orange, 12-pixel Times New Roman text. Today's date background color is grey, and is set directly in the `mx:DateChooser` tag.

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        .myHeaderStyle{
            color:#6666CC;
            font-family:Times New Roman, Times, serif;
            font-size:16px; font-weight:bold;}
        .myTodayStyle{
            color:#CC6633;
            font-family:Times New Roman, Times, serif;
            font-size:12px; font-weight:bold;}
        .myDayStyle{
            color:#006600;
            font-family:Courier New, Courier, mono;
            font-size:15px; font-style:italic; font-weight:bold;}
    </mx:Style>

    <mx:DateChooser
        headerStyleName="myHeaderStyle"
        todayStyleName="myTodayStyle"
        todayColor="#CCCCCC"
        weekDayStyleName="myDayStyle"/>
</mx:Application>
```

Specifying selectable dates

The DateChooser control has the following properties that let you specify which dates a user can select:

Property	Description
`disabledDays`	An array of days of the week that the user cannot select. Often used to disable weekend days.
`disabledRange`	An array of dates that the user cannot select. The array can contain individual Date objects, objects specifying date ranges, or both.
`selectableRange`	A single range of dates that the user can select. The user can navigate only among the months that include this range; in these months any dates outside the range are disabled. Use the `disabledRange` property to disable dates within the selectable range.

The following example shows a DateChooser control that has the following characteristics:

- The `selectableRange` property limits users to selecting dates in the range January 1 - March 15, 2006. Users can only navigate among the months of January through March 2006.
- The `disabledRanges` property prevents users from selecting January 11 or any day in the range January 23 - February 10.
- The `disabledDays` property prevents users from selecting Saturdays or Sundays.

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:DateChooser
    selectableRange="{{rangeStart: new Date(2006,0,1),
        rangeEnd: new Date(2006,2,15)}}"
    disabledRanges="{[new Date(2006,0,11),
        {rangeStart: new Date(2006,0,23), rangeEnd: new Date(2006,1,10)}]}"
    disabledDays="{[0,6]}"/>
</mx:Application>
```

Setting DateChooser and DateField properties in ActionScript

Properties of the DateChooser and DateField controls take values that are scalars, Arrays, and Date objects. While you can set most of these properties in MXML, it can be easier to set some in ActionScript.

For example, the following code example uses an array to set the `disabledDays` property so that Saturday and Sunday are disabled, which means that they cannot be selected in the calendar. This example sets the `disabledDays` property in two different ways: using tags and using tag attributes:

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserDisabledOption.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Use tags.-->
    <mx:DateField>
            <mx:disabledDays>
                <mx:Number>0</mx:Number>
                <mx:Number>6</mx:Number>
            </mx:disabledDays>
    </mx:DateField>

    <!-- Use tag attributes.-->
    <mx:DateField disabledDays="[0,6]"/>
</mx:Application>
```

The following example sets the `dayNames`, `firstDayOfWeek`, `headerColor`, and `selectableRange` properties of a DateChooser control using an `initialize` event:

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserInitializeEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.events.DateChooserEvent;

            private function dateChooser_init():void {
                myDC.dayNames=['Sun', 'Mon', 'Tue',
                    'Wed', 'Th', 'Fri', 'Sat'];
                myDC.firstDayOfWeek = 3;
                myDC.setStyle("headerColor", 0xff0000);
                myDC.selectableRange = {rangeStart: new Date(2004,0,1),
                    rangeEnd: new Date(2007,0,10)};
            }

            private function onScroll():void {
                myDC.setStyle("fontStyle", "italic");
            }
        ]]>
    </mx:Script>

    <mx:DateChooser id="myDC"
        width="200"
        creationComplete="dateChooser_init();"
        scroll="onScroll();"/>
</mx:Application>
```

To set the `selectableRange` property, the code creates two Date objects that represent the first date and last date of the range. Users can only select dates within the specified range. This example also changes the `fontStyle` of the DateChooser control to `italics` after the first time the user scrolls it.

Formatting dates with the DateField control

You can use the `formatString` property of the DateField control to format the string in the control's text field. The `formatString` property can contain any combination of "MM", "DD", "YY", "YYYY", delimiter, and punctuation characters. The default value is "MM/DD/YYYY".

In the following example, you set the `formatString` property to "MM/DD/YY" to display a two-digit year:

```
<?xml version="1.0"?>
<!-- controls\date\DateFieldFormat.mxml -->
<mx:Application  xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Label text="{date2.formatString}"/>
    <mx:DateField id="date2"
        editable="true"
        width="100"
        formatString="MM/DD/YY"/>
</mx:Application>
```

The DateField control also lets you specify a formatter function that converts the date to a string in your preferred format for display in the control's text field. The DateField `labelFunction` property and the DateFormatter class help you format dates.

By default, the date in the DateField control text field is formatted in the form "MM/DD/YYYY". You use the `labelFunction` property of the DateField control to specify a function to format the date displayed in the text field, and return a String containing the date. The function has the following signature:

```
public function formatDate(currentDate:Date):String {
  ...
  return dateString;
}
```

You can choose a different name for the function, but it must take a single argument of type Date and return the date as a String for display in the text field. The following example defines the function `formatDate()` to display the date in the form yyyy/mm/dd, such as 2005/11/24. This function uses a DateFormatter object to do the formatting.

```
<?xml version="1.0"?>
<!-- controls\date\DateChooserFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function formatDate(date:Date):String {
                return dfconv.format(date);
            }
        ]]>
    </mx:Script>

    <mx:DateFormatter id="dfconv" formatString="YYYY/MM/DD"/>
    <mx:DateField id="df" labelFunction="formatDate" parseFunction="null"/>
</mx:Application>
```

The `parseFunction` property specifies a function that parses the date entered as text in the text field of the DateField control and returns a Date object to the control. If you do not allow the user to enter a date in the text field, set the `parseFunction` property to `null` when you set the `labelFunction` property.

If you want to let the user enter a date in the control's text field, you should specify a function to the `parseFunction` property that converts the text string to a Date object for use by the DateField control. If you set the `parseFunction` property, it should typically perform the reverse of the function specified to the `labelFunction` property.

The function specified to the `parseFunction` property has the following signature:

```
public function parseDate(valueString:String, inputFormat:String):Date {
   ...
   return newDate
}
```

Where the `valueString` argument contains the text string entered by the user in the text field, and the `inputFormat` argument contains the format of the string. For example, if you only allow the user to enter a text sting using two characters for month, day, and year, then pass "MM/DD/YY" to the `inputFormat` argument.

User interaction

The date chooser includes arrow buttons that let users move between months. Users can select a date with the mouse by clicking the desired date.

Clicking a forward month arrow advances a month; clicking the back arrow displays the previous month. Clicking forward a month on December, or back on January, moves to the next (or previous) year. Clicking a date selects it. By default, the selected date is indicated by a green background around the date and the current day is indicated by a black background with the date in white. Clicking the currently selected date deselects it.

The following keystrokes let users navigate DateChooser and DateField control:

Key	Use
Left Arrow	Moves the selected date to the previous enabled day in the month. Does not move to the previous month.
Right Arrow	Moves the selected date to the next enabled day in the month. Does not move to the next month.
Up Arrow	Moves the selected date up the current day of week column to the previous enabled day. Does not move to the previous month.
Down Arrow	Moves the selected date down the current day of week column to next enabled day. Does not move to the next month.

Key	Use
Page Up	Displays the calendar for the previous month.
Page Down	Displays the calendar for the next month.
Home	Moves the selection to the first enabled day of the month.
End	Moves the selection to the last enabled day of the month.
+	Move to the next year.
-	Move to the previous year.
Control+Down Arrow	DateField only: open the DateChooser control.
Control+Up Arrow	DateField only: close the DateChooser control.
Escape	DateField only: cancel operation.
Enter	DateField only: selects the date and closes the DateChooser control.

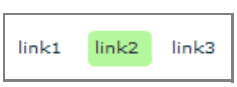
NOTE

The user must select the control before using navigation keystrokes. In a DateField control, all listed keystrokes work only when the date chooser is displayed.

LinkButton control

The LinkButton control creates a single-line hypertext link that supports an optional icon. You can use a LinkButton control to open a URL in a web browser.

The following example shows three LinkButton controls:



The LinkButton control has the following default properties:

Property	Default value
Default size	Width and height large enough for the text
Minimum size	0
Maximum size	Undefined

Creating a LinkButton control

You define a LinkButton control in MXML using the `<mx:LinkButton>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\button\LBSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:HBox>
        <mx:LinkButton label="link1"/>
        <mx:LinkButton label="link2"/>
        <mx:LinkButton label="link3"/>
    </mx:HBox>
</mx:Application>
```

The following code contains a single LinkButton control that opens a URL in a web browser window:

```
<?xml version="1.0"?>
<!-- controls\button\LBSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:LinkButton label="ADBE"
        width="100"
        click="navigateToURL(new URLRequest('http://quote.yahoo.com/
q?s=ADBE'), 'quote')"/>
</mx:Application>
```

This example uses the `navigateToURL()` method to open the URL.

The LinkButton control automatically provides visual cues when you move your mouse pointer over or click the control. The previous code example contains no link handling logic but does change color when you move your mouse pointer over or click a link.

The following code example contains LinkButton controls for navigating in a ViewStack navigator container:

```
<?xml version="1.0"?>
<!-- controls\button\LBViewStack.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>

        <!-- Put the links in an HBox container across the top. -->
        <mx:HBox>
            <mx:LinkButton label="Link1"
                click="viewStack.selectedIndex=0;"/>
            <mx:LinkButton label="Link2"
                click="viewStack.selectedIndex=1;"/>
            <mx:LinkButton label="Link3"
                click="viewStack.selectedIndex=2;"/>
        </mx:HBox>

        <!-- This ViewStack container has three children. -->
        <mx:ViewStack id="viewStack">
            <mx:VBox width="150">
                <mx:Label text="One"/>
            </mx:VBox>
            <mx:VBox width="150">
                <mx:Label text="Two"/>
            </mx:VBox>
            <mx:VBox width="150">
                <mx:Label text="Three"/>
            </mx:VBox>
        </mx:ViewStack>
    </mx:VBox>
</mx:Application>
```

A LinkButton control's label is centered within the bounds of the LinkButton control. You can position the text label in relation to the icon using the `labelPlacement` property, which accepts the values `right`, `left`, `bottom`, and `top`.

LinkButton control user interaction

When a user clicks a LinkButton control, the LinkButton control dispatches a `click` event. If a LinkButton control is enabled, the following happens:

- When the user moves the mouse pointer over the LinkButton control, the LinkButton control changes its rollover appearance.
- When the user clicks the LinkButton control, the input focus moves to the control and the LinkButton control displays its pressed appearance. When the user releases the mouse button, the LinkButton control returns to its rollover appearance.
- If the user moves the mouse pointer off the LinkButton control while pressing the mouse button, the control's appearance returns to its original state and the control retains input focus.
- If the toggle property is set to `true`, the state of the LinkButton control does not change until the mouse button is released over the control.

If a LinkButton control is disabled, it appears as disabled, regardless of user interaction. In the disabled state, the control ignores all mouse or keyboard interaction.

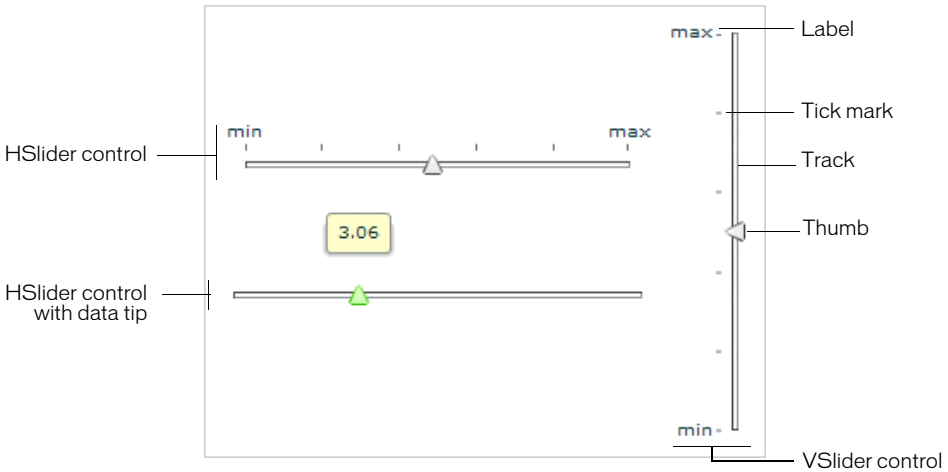
HSlider and VSlider controls

You can use the slider controls to select a value by moving a slider thumb between the end points of the slider track. The current value of the slider is determined by the relative location of the thumb between the end points of the slider, corresponding to the slider's minimum and maximum values.

By default, the minimum value of a slider is 0 and the maximum value is 10. The current value of the slider can be any value in a continuous range between the minimum and maximum values, or it can be one of a set of discrete values, depending on how you configure the control.

About Slider controls

Flex provides two sliders: the HSlider (Horizontal Slider) control, which creates a horizontal slider, and the VSlider (Vertical Slider) control, which creates a vertical slider. The following example shows the HSlider and VSlider controls:



This example includes the data tip, slider thumb, track, tick marks, and labels. You can optionally show or hide data tips, tick marks, and labels.

The following code example reproduces this image (without annotations):

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox>
        <mx:VBox>
            <mx:HSlider
                tickInterval="2"
                labels="['min', 'max']" height="150"/>
            <mx:HSlider/>
        </mx:VBox>

        <mx:VSlider
            tickInterval="2"
            labels="['min', 'max']"/>
    </mx:HBox>
</mx:Application>
```

The HSlider and VSlider controls have the following default properties:

Property	Default value
Default size	**Horizontal Slider** 250 pixels wide, and high enough to hold the slider and any associated labels **Vertical Slider** 250 pixels high, and wide enough to hold the slider and any associated labels
Minimum size	None
Maximum size	None

Creating a Slider control

You define an HSlider control in MXML using the `<mx:HSlider>` tag and a VSlider control using the `<mx:VSlider>` tag. You must specify an `id` value if you intend to refer to a component elsewhere, either in another tag or in an ActionScript block.

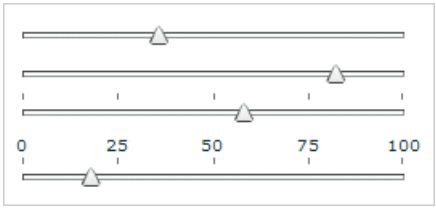
The following code example creates four HSlider controls:

- The first slider has a maximum value of 100, and lets the user move the slider thumb to select a value in the continuous range between 0 and 100.
- The second slider uses the `snapInterval` property to define the discrete values between the minimum and maximum that the user can select. In this example, the `snapInterval` is 5, which means that the user can select the values 0, 5, 10, 15, and so on.
- The third slider uses the `tickInterval` property to add tick marks and set the interval between the tick marks to 25, so that Flex displays a tick mark along the slider corresponding to the values 0, 25, 50, 75, and 100. Flex displays tick marks whenever you set the `tickInterval` property to a nonzero value.

- The fourth slider uses the `labels` property to add labels and set them at each tick mark. The labels property accepts an array of values to display. It automatically distributes them evenly along the slider. The first value always corresponds to the leftmost edge of the slider and the last value always corresponds to the rightmost edge of the slider.

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <mx:HSlider
            maximum="100"/>
        <mx:HSlider
            maximum="100"
            snapInterval="5"/>
        <mx:HSlider
            maximum="100"
            snapInterval="5"
            tickInterval="25"/>
        <mx:HSlider
            maximum="100"
            snapInterval="5"
            tickInterval="25"
            labels="[0,25,50,75,100]"/>
    </mx:VBox>
</mx:Application>
```

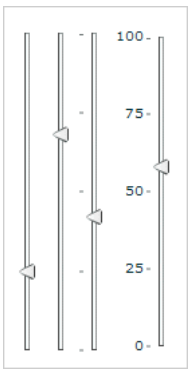
This example produces the following image:



You can do a similar thing by using `VSlider` controls:

```
<?xml version="1.0"?>
<!-- controls\slider\VSliderSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox>
        <mx:VSlider
            maximum="100"/>
        <mx:VSlider
            maximum="100"
            snapInterval="5"/>
        <mx:VSlider
            maximum="100"
            snapInterval="5"
            tickInterval="25"/>
        <mx:VSlider
            maximum="100"
            snapInterval="5"
            tickInterval="25"
            labels="[0,25,50,75,100]"/>
    </mx:HBox>
</mx:Application>
```

This code results in the following application:



You can bind the `value` property of a slider to another control to display the current value of the slider. The following example binds the `value` property to a Text control:

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HSlider id="mySlider" maximum="100"/>
    <mx:Text text="{mySlider.value}"/>
</mx:Application>
```

This code produces the following image:

Using slider events

The slider controls let the user select a value by moving the slider thumb between the minimum and maximum values of the slider. You use an event with the slider to recognize when the user has moved the thumb, and to determine the current value associated with the slider.

The slider controls can dispatch the events described in the following table:

Event	Description
`change`	Dispatches when the user moves the thumb. If the `liveDragging` property is `true`, the event is dispatched continuously as the user moves the thumb. If `liveDragging` is `false`, the event is dispatched when the user releases the slider thumb.
`thumbDrag`	Dispatches when the user moves a thumb.
`thumbPress`	Dispatches when the user selects a thumb using the mouse pointer.
`thumbRelease`	Dispatches when the user releases the mouse pointer after a `thumbPress` event occurs.

The following code example uses a `change` event to show the current value of the slider in a TextArea control when the user releases the slider thumb:

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.events.SliderEvent;
            import mx.controls.sliderClasses.Slider;

            private function sliderChange(event:SliderEvent):void {
                var currentSlider:Slider=Slider(event.currentTarget);
                thumb.text=String(currentSlider.value);
            }
        ]]>
    </mx:Script>

    <mx:HSlider change="sliderChange(event);"/>
    <mx:TextArea id="thumb"/>
</mx:Application>
```

By default, the `liveDragging` property of the slider control is set to `false`, which means that the control dispatches the `change` event when the user releases the slider thumb. If you set `liveDragging` to `true`, the control dispatches the `change` event continuously as the user moves the thumb, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderEventLiveDrag.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.events.SliderEvent;
            import mx.controls.sliderClasses.Slider;

            private function sliderChangeLive(event:SliderEvent):void {
                var currentSlider:Slider=Slider(event.currentTarget);
                thumbLive.text=String(currentSlider.value);
            }
        ]]>
    </mx:Script>

    <mx:HSlider
        liveDragging="true"
        change="sliderChangeLive(event);"/>
    <mx:TextArea id="thumbLive"/>
</mx:Application>
```

Using multiple thumbs

You can configure a slider control to have one thumb, or two thumbs. If you configure the slider to use a single thumb, you can move the thumb anywhere between the end points of the slider. If you configure it to have two thumbs, you cannot drag one thumb across the other thumb.

When you configure a slider control to have two thumbs, you use the `values` property of the control to access the current value of each thumb. The `values` property is a two-element array that contains the current value of the thumbs, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderMultThumb.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.events.SliderEvent;
            import mx.controls.sliderClasses.Slider;

            private function sliderChangeTwo(event:SliderEvent):void {
                var ct:Slider=Slider(event.currentTarget);
                thumbTwoA.text=String(ct.values[0]);
                thumbTwoB.text=String(ct.values[1]);
                thumbIndex.text=String(event.thumbIndex);
            }
        ]]>
    </mx:Script>

    <mx:HSlider thumbCount="2"
        change="sliderChangeTwo(event);"/>
    <mx:TextArea id="thumbTwoA"/>
    <mx:TextArea id="thumbTwoB"/>
    <mx:TextArea id="thumbIndex"/>
</mx:Application>
```

This example also uses the `thumbIndex` property of the event object. This property has a value of 0 if the user modified the position of the first thumb, and a value of 1 if the user modified the position of the second thumb.

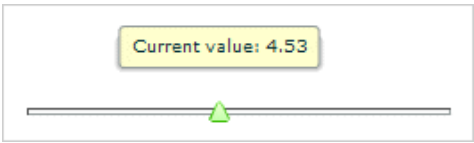
Using data tips

By default, when you select a slider thumb, a data tip appears, showing the current value of the slider. As you move the selected thumb, the data tip shows the new slider value. You can disable data tips by setting the `showDataTip` property to `false`.

You can use the `dataTipFormatFunction` property to specify a callback function to format the data tip text. This function takes a single String argument containing the data tip text, and returns a String containing the new data tip text, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\slider\HSliderDataTip -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function myDataTipFunc(val:String):String {
                return "Current value: " + String(val);
            }
        ]]>
    </mx:Script>

    <mx:HSlider
        height="80"
        dataTipFormatFunction="myDataTipFunc"/>
</mx:Application>
```

This code produces the following image:



In this example, the data tip function prepends the data tip text with the String "Current value: ". You can modify this example to insert a dollar sign ($) prefix on the data tip for a slider that controls the price of an item.

Keyboard navigation

The HSlider and VSlider controls have the following keyboard navigation features when the slider control has focus:

Key	Description
Left Arrow	Decrement the value of an HSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel.
Right Arrow	Increment the value of a HSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel.
Home	Moves the thumb of an HSlider control to its minimum value.
End	Moves the thumb of an HSlider control to its maximum value.

Key	Description
Up Arrow	Increment the value of an VSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel.
Down Arrow	Decrement the value of a VSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel.
Page Down	Moves the thumb of a VSlider control to its minimum value.
Page Up	Moves the thumb of a VSlider control to its maximum value.

SWFLoader control

The SWFLoader control lets you load one Flex 2 application into another Flex application. It has properties that let you scale its contents. It can also resize itself to fit the size of its contents. By default, content is scaled to fit the size of the SWFLoader control. The SWFLoader control can also load content on demand programmatically, and monitor the progress of a load operation.

The SWFLoader control also lets you load the contents of a GIF, JPEG, PNG, SVG, or SWF file into your application, where the SWF file does not contain a Flex 2 application.

NOTE

Flex also includes the Image control for loading GIF, JPEG, PNG, SVG, or SWF files. You typically use the Image control for loading static graphic files and SWF files, and use the SWFLoader control for loading Flex 2 applications as SWF files. The Image control is also designed to be used in custom cell renderers and item editors.
For more information on the Image control, see "Image control" on page 325. For more information on using the SWFLoader control to load a Flex application, see "Using the SWFLoader control to load a Flex Data Services application" on page 323.

A SWFLoader control cannot receive focus. However, content loaded into the SWFLoader control can accept focus and have its own focus interactions.

The SWFLoader control has the following default properties:

Property	Default value
Default size	Width and height large enough for the loaded content
Minimum size	0
Maximum size	Undefined

Creating a SWFLoader control

You define a SWFLoader control in MXML using the `<mx:SWFLoader>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\swfloader\SWFLoaderSimple.mxml-->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:SWFLoader id="loader1" source="FlexApp.swf"/>
</mx:Application>
```

Like the Image control, you can also use the Embed statement with the SWFLoader control to embed the image in your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\swfloader\SWFLoaderSimpleEmbed.mxml-->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:SWFLoader id="loader1" source="@Embed(source='flexapp.swf')"/>
</mx:Application>
```

When using the SWFLoader control with an SVG file, you can only load it using an Embed statement; you cannot load an SVG file at run time. For more information about embedding resources, see the description for the Image control at "About importing images" on page 325, and Chapter 30, "Embedding Assets," on page 1113.

This technique works well with SWF files that add graphics or animations to an application, but are not intended to have a large amount of user interaction. If you import SWF files that require a large amount of user interaction, you should build them as custom components. For more information on custom components, see *Creating and Extending Flex 2 Components*.

Interacting with a loaded Flex 2 application

The following example, in the file FlexApp.mxml, shows a simple Flex application that defines two Label controls, a variable, and a method to modify the variable:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="200" width="200">

    <mx:Script>
        <![CDATA[
            [Bindable]
            public var varOne:String = "This is a public variable";

            public function setVarOne(newText:String):void {
                varOne=newText;
            }
        ]]>
    </mx:Script>

    <mx:Label id="lblOne" text="I am here"/>
    <mx:Label text="{varOne}"/>

    <mx:Button label="Nested Button" click="setVarOne('Nested button
pressed.');"/>

</mx:Application>
```

You compile this example into the file FlexApp.SWF, and then use the SWFLoader control to load it into another Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\swfloader\SWFLoaderInteract.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.managers.SystemManager;
            import mx.controls.Label;

            [Bindable]
            public var loadedSM:SystemManager;

            // Initialize variables with information from
            // the loaded application.
            private function initNestedAppProps():void {
                loadedSM = SystemManager(myLoader.content);
            }

            // Update the Label control in the outer application
            // from the Label control in the loaded application.
            public function updateLabel():void {
                lbl.text=loadedSM.application["lblOne"].text;
            }

            // Write to the Label control in the loaded application.
            public function updateNestedLabels():void {
                loadedSM.application["lblOne"].text = "I was just updated";
                loadedSM.application["varOne"] = "I was just updated";
            }

            // Write to the varOne variable in the loaded application
            // using the setVarOne() method of the loaded application.
            public function updateNestedVarOne():void {
                FlexApp(loadedSM.application).setVarOne("Updated varOne!");
            }
        ]]>
    </mx:Script>

    <mx:Label id="lbl"/>
    <mx:SWFLoader id="myLoader" width="300"
        source="FlexApp.swf"
        creationComplete="initNestedAppProps();"/>

    <mx:Button label="Update Label Control in Outer Application"
        click="updateLabel();"/>
    <mx:Button label="Update Nested Controls"
```

```
        click="updateNestedLabels();"/>
    <mx:Button label="Update Nexted varOne"
        click="updateNestedVarOne();"/>
</mx:Application>
```

Notice that this application loads the SWF file at run time, it does not embed it. For information on embedding a Flex 2 application using the SWFLoader tag, see Chapter 30, "Embedding Assets," on page 1113.

In the preceding example, you use the `creationComplete` event of the SWFLoader control to initialize two variables; the first contains a reference to the SystemManager object for the loaded Flex application, and the second contains a reference to the Label control in the loaded application.

When a user clicks the first Button control in the outer application, Flex copies the text from the Label control in the loaded application to the Label control in the outer application.

When a user clicks the second Button control, Flex writes the text to the Label control and to the *`varOne`* variable defined in the loaded application.

When a user clicks the third Button control, Flex uses the `setVarOne()` method of the loaded application to write to the *`varOne`* variable defined in the loaded application.

Using the SWFLoader control to load a Flex Data Services application

Flex Data Services users can use the SWFLoader control to load a Flex application. The following code example loads the file buttonicon.mxml, where buttonicon.mxml is the example found in "Embedding an icon in a Button control" on page 270:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:SWFLoader source="buttonicon.mxml.swf" scaleContent="false"/>
</mx:Application>
```

In this example, you specify the value of the source property as `"buttonicon.mxml.swf"`. Adobe Flex Data Services compiles the file buttonicon.mxml, and returns the SWF file to the main application. If you had specified the value as `"buttonicon.swf"`, Flex Data Services returns the SWF file if it exists, but will not compile buttonicon.mxml if it does not.

Externalizing application classes

To reduce the size of the applications that you load using the SWFLoader control, you can instruct the loaded application to externalize framework classes that are also included by the loading application. The result is that the loaded application is smaller because it only includes the classes it requires, while the framework code and other dependencies are included in the loading application.

To externalize framework classes, you generate a linker report from the loading application by using `link-report` option to the mxmlc command. You then use the `load-externs` option to the mxmlc compiler to specify this report when you compile the loaded application.

To externalize framework classes:

1. Generate the linker report for the loading application:

   ```
   mxmlc -link-report=report.xml MyApplication.mxml
   ```

2. Compile the loaded application using the link report:

   ```
   mxmlc -load-externs=report.xml MyLoadedApplication.mxml
   ```

3. Compile the loading application:

   ```
   mxmlc MyApplication.mxml
   ```

NOTE	If you externalize the loaded application's dependencies by using the `load-externs` option, your loaded application might not be compatible with future versions of Adobe Flex. Therefore, you might be required to recompile the application. To ensure that a future Flex application can load you application, compile that module with all the classes it requires.

For more information, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Sizing a SWFLoader control

You use the SWFLoader control's `scaleContent` property to control the sizing behavior of the SWFLoader control. When the `scaleContent` property is set to `true`, Flex scales the content to fit within the bounds of the control. However, images will still retain their aspect ratio by default.

Image control

Adobe Flex supports several image formats, including GIF, JPEG, PNG, SVG, and SWF files. You can import these images into your applications by using the Image control.

NOTE

Flex also includes the SWFLoader control for loading Flex 2 applications. You typically use the Image control for loading static graphic files and SWF files, and use the SWFLoader control for loading Flex 2 applications. The Image control is also designed to be used in custom item renderers and item editors. For more information on the SWFLoader control, see "SWFLoader control" on page 319.

About importing images

Flex supports importing GIF, JPEG, PNG, and SWF files at run time, and embedding GIF, JPEG, PNG, SVG, and SWF at compile time. The method you choose depends on the file types of your images and your application parameters.

Embedded images load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded images also require you to recompile your applications whenever your image files change. For an overview of resource embedding, see Chapter 30, "Embedding Assets," on page 1113.

The alternative to embedding a resource is to load the resource at run time. You can load a resource from the local file system in which the SWF file runs, or you can access a remote resource, typically though an HTTP request over a network. These images are independent of your Flex application, so you can change them without causing a recompile operation as long as the names of the modified images remain the same. The referenced images add no additional overhead to an application's initial loading time. However, you might experience a delay when you use the images and load them into Adobe Flash Player.

A SWF file can access one type of external resource only, either local or over a network; it cannot access both types. You determine the type of access allowed by the SWF file using the `use-network` flag when you compile your application. When `use-network` flag is set to `false`, you can access resources in the local filesystem, but not over the network. The default value is `true`, which allows you to access resources over the network, but not in the local filesystem.

For more information on the `use-network` flag, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

When you load images at run time, you should be aware of the security restrictions of Flash Player. For example, you can reference an image by using a URL, but the default security settings only permit Flex applications to access resources stored on the same domain as your application. To access images on other servers, you must use a crossdomain.xml file.

For more information on application security, see Chapter 4, "Applying Flex Security," in *Building and Deploying Flex 2 Applications*.

SVG drawing restrictions

You should be aware of the following restrictions when working with SVG files in Flex:

- You can only embed an SVG file in an application; you cannot load one at run time
- SMIL and animation are not supported
- Masking and filters are not supported
- Pattern Fill and some advanced Gradients are not supported
- Interactivity and scripting is not supported
- SVG text is rendered as nonsearchable and nonselectable SWF shape outlines, meaning it is not rendered as native text in Flash Player

Controlling image importing with the Image control

The Image control supports the following actions when you import an image:

- Specifying the image path
- Sizing an image
- Positioning the image in a Canvas container
- Setting visibility

Specifying the image path

The value of the `source` property of an Image control specifies a relative or absolute path, or URL to the imported image file. If the value is relative, it is relative to the directory that contains the file that uses the tag. For more examples, see "Specifying the image path" on page 326.

The `source` property has the following forms:

- `source="@Embed(source='relativeOrAbsolutePath')"`

The referenced image is packaged within the generated SWF file at compile-time when Flex creates the SWF file for your application. You can embed GIF, JPEG, PNG, SVG, and SWF files. When you embed an image, the value of the `source` property must be a relative or absolute path to a file on your local file system; it cannot be a URL.

The following example embeds a JPEG image into a Flex application:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Image id="loader1" source="@Embed(source='logo.jpg')"/>
</mx:Application>
```

In this example, the size of the image is the default size of the image file.

- `source="relativeOrAbsolutePathOrURL"`

  Flex loads the referenced image file at run time; it is not packaged as part of the generated SWF file. You can only reference GIF, JPEG, PNG, and SWF files. When `use-network` flag is set to `false`, you can access resources in the local filesystem, but not over the network. The default value is `true`, which allows you to access resources over the network, but not in the local filesystem.

  The following example access a JPEG image at run time:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Image id="loader1" source="logo.jpg"/>
</mx:Application>
```

  Because you did not use `@Embed` in the `source` property, Flex loads the image at run time.

In many applications, you create a directory to hold your application images. Commonly, that directory is a subdirectory of your main application directory. The `source` property supports relative paths to images, which lets you specify the location of an image file relative to your application directory.

The following example stores all images in an assets subdirectory of the application directory:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleAssetsDir.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Image id="loader1" source="@Embed(source='assets/logo.jpg')"/>
</mx:Application>
```

The following example uses a relative path to reference an image in an assets directory at the same level as the application's root directory:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleAssetsDirTop.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Image id="loader1" source="@Embed(source='../assets/logo.jpg')"/>
</mx:Application>
```

You can also reference an image using a URL, but the default security settings only permit Flex applications to access resources stored on the same domain as your application. To access images on other servers, you must use a crossdomain.xml file.

The following example shows how to reference an image using a URL:

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleAssetsURL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Image id="image1"
        source="http://localhost:8100/flex/assets/logo.jpg"/>
</mx:Application>
```

NOTE

You can use relative URLs for images hosted on the same web server as the Flex application, but you must load these images over the Internet rather than access them locally.

Using an image multiple times

You can use the same image multiple times in your application by using the normal image import syntax each time. Flex only loads the image once, and then references the loaded image as many times as necessary.

Sizing an image

Flex sets the height and width of an imported image to the height and width settings in the image file. By default, Flex does not resize the image.

To set an explicit height or width for an imported image, set its `height` and `width` properties of the Image control. Setting the `height` or `width` property prevents the parent from resizing it. The `scaleContent` property has a default value of `true`, therefore, Flex scales the image as it resizes it to fit the specified height and width. The aspect ratio is maintained by default, so the image may not completely fill the designated space. Set the `scaleContent` property to `false` to disable scaling. Set the `maintainAspectRatio` property to `false` to allow an image to fill all available space regardless of its dimensions. For more information about image aspect ratios, see "Maintaining aspect ratio when sizing" on page 329.

To let Flex resize the image as part of laying out your application, set the `height` or `width` properties to a percentage value. Flex attempts to resize components with percentage values for these properties to the specified percentage of their parent container. You can also use the `maxHeight` and `maxWidth` and `minHeight` and `minWidth` properties to limit resizing. For more information on resizing, see Chapter 13, "Introducing Containers," on page 491.

One common use for resizing an image is to create image thumbnails. In the following example, the image has an original height and width of 100 by 100 pixels. By specifying a height and width of 20 by 20 pixels, you create a thumbnail of the image.

```
<?xml version="1.0"?>
<!-- controls\image\ImageSimpleThumbnail.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Image id="image1"
        source="logo.jpg"
        width="20" height="20"/>
    <mx:Image id="image2"
        source="logo.jpg"/>
</mx:Application>
```

Maintaining aspect ratio when sizing

The aspect ratio of an image is the ratio of its width to its height. For example, a standard NTSC television set uses an aspect ratio of 4:3, and an HDTV set uses an aspect ratio of 16:9. A computer monitor with a resolution of 640 by 480 pixels also has an aspect ratio of 4:3. A square has an aspect ratio of 1:1.

All images have an inherent aspect ratio. When you use the `height` and `width` properties of the Image control to resize an image, by default Flex preserves the aspect ratio of the image so that it does not appear distorted.

By preserving the aspect ratio of the image, Flex might not draw the image to fill the entire height and width specified for the `<mx:Image>` tag. For example, if your original image is a square 100 by 100 pixels, which means it has an aspect ratio of 1:1, and you use the following statement to load the image:

```
<mx:Image source="myImage.jpg" height="200" width="200"/>
```

The image increase to four times its original size and fills the entire 200 x 200 pixel area.

The following example sets the height and width of the same image to 150 by 200 pixels, an aspect ratio of 3:4:

```
<mx:Image source="myImage.jpg" height="150" width="200"/>
```

In this example, you do not specify a square area for the resized image. Flex maintains the aspect ratio of an image by default, therefore, Flex sizes the image to 150 by 150 pixels, the largest possible image that maintains the aspect ratio and conforms to the size constraints. The other 50 by 150 pixels remain empty. However, the `<mx:Image>` tag reserves the empty pixels and makes them unavailable to other controls and layout elements.

You can use a Resize effect to change the width and height of an image in response to a trigger. As part of configuring the Resize effect, you specify a new height and width for the image. Flex maintains the aspect ratio of the image by default, so it resizes the image as much as possible to conform to the new size, while maintaining the aspect ratio. For example, place your mouse pointer over the image in this example to enlarge it, and then move the mouse off the image to shrink it to its original size:

```
<?xml version="1.0"?>
<!-- controls\image\ImageResize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Resize id="resizeBig" widthFrom="120" widthTo="200"/>
    <mx:Resize id="resizeSmall" widthFrom="200" widthTo="120"/>

    <mx:Image width="120"
        source="@Embed('logo.jpg')"
        rollOverEffect="{resizeBig}"
        rollOutEffect="{resizeSmall}"/>
</mx:Application>
```

For more information on the Resize effect, see Chapter 17, “Using Behaviors,” on page 649.

If you do not want to preserve the aspect ratio when you resize an image, you can set the `maintainAspectRatio` property to `false`. By default, `maintainAspectRatio` is set to `true` to enable the preservation of the aspect ratio.

The following example resizes the Flex logo to the exact values of the height and width properties without regard for its aspect ratio:

```
<?xml version="1.0"?>
<!-- controls\image\ImageResizeMaintainAR.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Image id="image1"
        source="@Embed('logo.jpg')"
        width="250" height="100"
        maintainAspectRatio="false"/>
</mx:Application>
```

By choosing not to maintain the aspect ratio, you allow for the possibility of a distorted image. For example, the Adobe logo is 136 by 47 pixels by default. In the following example, it is distorted because the aspect ratio is not maintained when the image is resized:

Positioning the image in a Canvas container

A Canvas container, and the Panel and Application containers with the `layout` property set to `absolute`, let you specify the location of its children within the container. To specify the absolute position of an image, you use the `x` and `y` properties of the Image control.

NOTE In all other containers, the container controls the positioning of its children and ignores the `x` and `y` properties.

The `x` and `y` properties specify the location of the upper-left corner of the image in the Canvas container. In the following example, you set the position of the image at (40,40), 40 pixels down and 40 pixels to the right of the upper-left corner of the Canvas container:

```
<?xml version="1.0"?>
<!-- controls\image\ImageCanvas.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Canvas id="canvas0"
        borderStyle="solid"
        width="200"
        height="200">
        <mx:Image id="img0"
            source="@Embed('logo.jpg')"
            x="40" y="40"/>
    </mx:Canvas>
</mx:Application>
```

This code results in the following application:

Setting visibility

The `visible` property of the Image control lets you load an image but render it invisible. By default, the image is visible. To make an image invisible, set the `visible` property to `false`. The following example loads an image but does not make it visible:

```
<?xml version="1.0"?>
<!-- controls\image\ImageVisible.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox id="vbox0"
        borderStyle="solid">
        <mx:Image id="img0"
            visible="false"
            source="@Embed(source='logo.jpg')"/>
    </mx:VBox>
</mx:Application>
```

The VBox container still allocates space for the image when it lays out its children. Thus, the VBox is the same size as the image file and, if your application contained a Button control after the `<mx:Image>` tag, the button would appear in the same location as if the image was visible.

If you want to make the image invisible, and have its parent container ignore the image when sizing and positioning its other children, set the `includeInLayout` property of the Image control to `false`. By default, the `includeInLayout` property to `true`, so that even when the image is invisible, the container sizes and positions it as if it were visible.

Often, you use the `visible` property to mark all images invisible, except one image. For example, assume that you have an area of your application dedicated to showing one of three possible images based on some user action. You set the `visible` property set to `true` for only one of the possible images; you set the `visible` property set to `false` for all other images, which makes them invisible.

You can use ActionScript to set image properties. In the following example, when the user clicks the button, the action sets the `visible` property of the image to `true` to make it appear:

```
<?xml version="1.0"?>
<!-- controls\image\ImageAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function showImage():void {
                image1.visible=true;
            }
        ]]>
    </mx:Script>
    <mx:VBox id="vbox0"
        width="80"
        height="80">
        <mx:Image id="image1"
            visible="false"
            source="@Embed(source='logo.jpg')"/>
        <mx:Button id="myButton"
            label="Show" click="showImage();"/>
    </mx:VBox>
</mx:Application>
```

This following examples shows the results, before and after a user clicks the button:

If you set the `visible` property, you can control when images are loaded. By allocating space but making images invisible when a page loads, you ensure that any slower performance occurs during the initialization stage when users expect it, rather than as they interact with the application and perform actions that require the image. By setting the `visible` property, you can also prevent resizing and relayout within your application at seemingly random intervals.

Techniques for using the Image control

Often in a product catalog, when a user clicks an item, the catalog displays an image of the item. One strategy for building a catalog is to load the catalog images into your application, but make them all invisible. When a user selects a product, you make that image visible.

However, this strategy requires that you add an Image control for all the images in your catalog and load the images, even if they are invisible, when the application starts. The resulting SWF file would be unnecessarily large, because it would contain all the images and the start-up time would be negatively affected by loading invisible images.

A better strategy is to dynamically load the images from your server, as necessary. In this way, your SWF file stays small, because it does not have to contain invisible images, and your start-up time improves.

As part of its implementation, the ActionScript class that defines the `<mx:Image>` tag is a subclass of the SWFLoader class. After creating an image, you can use the properties and methods of the SWFLoader control with your image, including the `load()` method, which loads an image file dynamically.

NOTE The `load()` method of the SWFLoader control only works with GIF, JPEG, PNG, and SWF files; you cannot use it to load SVG files.

The following example uses the `load()` method to replace the logo.jpg image with the logowithtext.gif image when the user clicks a button:

```
<?xml version="1.0"?>
<!-- controls\image\ImageLoad.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      private function afterImage():void {
        image1.load('http://localhost:8100/flex/assets/logowithtext.jpg');
      }
    ]]>
  </mx:Script>

  <mx:VBox id="vbox0"
      width="150" height="100">
      <mx:Image id="image1" source="logo.jpg"/>
      <mx:Button id="myButton"
          label="Show Second Image"
          click="afterImage();"/>
  </mx:VBox>
</mx:Application>
```

Notice in this example that the application accesses the images in the local file system; they are not embedded or accessed by URL. Therefore, you have to compile this application with the `use-network` flag set to `false`.

The container that holds the image does not adjust the layout of its children when you call the `load()` method. Therefore, you typically replace one image with another image of the same size. If the new image is significantly larger than the original, it can overlay other components in the container.

You can make the selection of the replacement image based on a user action in your application. For example, you might want to load an image based on a user selection in a list box or data grid.

In the next example, you use the index number of the selected item in a data grid to determine the image to load. In this example, images are named 1.jpg, 2.jpg, 3.jpg, and so on, corresponding to items in the grid.

```
// Retrieve the image associated with the item selected in the grid.
private function getImage():void {
  var cartGrid = dgrid;
  var imageSource:String = 'images/' + cartGrid.getSelectedIndex() +
  '.jpg';
  image1.load(imageSource);
}
```

In this example, the images are stored in the images directory. The complete path to an image is the directory name, the index number, and the file suffix .jpg.

You register this function as the event listener for a `change` event in the data grid, as follows:

```
<mx:DataGrid id="dgrid" height="200" width="350" change="getImage();"/>
```

When a user changes the currently selected item in the data grid, Flex calls the `getImage()` function to update the displayed image.

You could modify this example to use information about the selected item to determine the image to load, rather than using the selected item's index. For example, the grid could contain a list of objects, where each object has a property that contains the image name associated with it.

In addition to the `load()` method, you can also access other properties of the SWFLoader control, including `percentLoaded`. The `percentLoaded` property is particularly useful because it lets you to display a progress bar so users know that the application did not become unresponsive. For a complete list of the SWFLoader properties and methods, see *Adobe Flex 2 Language Reference*.

VideoDisplay control

Flex supports the VideoDisplay control to incorporate streaming media into Flex applications. Flex supports the Flash Video File (FLV) file format with this control. This section describes how to use the VideoDisplay control in your application.

Using media in Flex

Media, such as movie and audio clips, are used more and more to provide information to web users. As a result, you need to provide users with a way to stream the media, and then control it. The following examples are usage scenarios for media controls:

- Showing a video message from the CEO of your company
- Streaming movies or movie previews
- Streaming songs or song snippets
- Providing learning material in the form of media

The Flex streaming VideoDisplay control makes it easy to incorporate streaming media into Flash presentations. Flex supports the Flash Video File (FLV) file format with this control. You can use this control with video and audio data. When you use the VideoDisplay control by itself your application provides no mechanism for its users to control the media files.

NOTE	The VideoDisplay control does not support scan forward and scan backward functionality. Also, the VideoDisplay control does not support accessibility or styles.

About the VideoDisplay control

Flex creates a VideoDisplay control with no visible user interface. It is simply a control to hold and play media.

NOTE	The user cannot see anything unless some video media is playing.

The `playheadTime` property of the control holds the current position of the playhead in the video file, measured in seconds. Most events dispatched by the control include the playhead position in the associated event object. One use of the playhead position is to dispatch an event when the video file reaches a specific position. For more information, see "Adding a cue point" on page 338.

The VideoDisplay control also supports the `volume` property. This property takes a value from 0.0 to 1.00; 0.0 is mute and 1.00 is the maximum volume. The default value is 0.75.

Setting the size of a media component

The appearance of any video media playing in a VideoDisplay control is affected by the following properties:

- `maintainAspectRatio`
- `height`

- `width`

When you set `maintainAspectRatio` to `true` (the default), the control adjusts the size of the playing media after the control size has been set. The size of the media is set to maintain its aspect ratio.

If you omit both `width` and `height` properties for the control, Flex makes the control the size of the playing media. If you specify only one property, and the `maintainAspectRatio` property is `false`, the size of the playing media determines the value of the other property. If the `maintainAspectRatio` property is `true`, the media retains its aspect ratio when resizing.

The following example creates a VideoDisplay control:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplaySimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:VBox>
        <mx:VideoDisplay
            source="http://localhost:8100/flex/assets/MyVideo.flv"
            height="400"
            width="400"/>
    </mx:VBox>
</mx:Application>
```

By default, Flex sizes the VideoDisplay control to the size of the media. If you specify the `width` or `height` property of the control, and either is smaller than the media's dimensions, Flex does not change the size of the component. Instead, Flex sizes the media to fit within the component. If the control's playback area is smaller than the default size of the media, Flex shrinks the media to fit inside the control.

Using methods of the VideoDisplay control

You can use the following methods of the VideoDisplay control in your application: `close()`, `load()`, `pause()`, `play()`, and `stop()`. The following example uses the `pause()` and `play()` methods in event listener for two Button controls to pause or play an FLV file:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayStopPlay.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <mx:VideoDisplay id="myVid"
            source="http://localhost:8100/flex/assets/MyVideo.flv"/>
        <mx:Button label="Pause" click="myVid.pause();"/>
        <mx:Button label="Play" click="myVid.play();"/>
    </mx:VBox>
</mx:Application>
```

Adding a cue point

You can use cue points to trigger events when the playback of your media reaches a specified location. To use cue points, you set the `cuePointManagerClass` property to mx.controls.videoClasses.CuePointManager to enable cue point management, and then pass an Array of cue points to the `cuePoints` property of the VideoDisplay control. Each element of the Array contains two fields. The `name` field contains an arbitrary name of the cue point. The `time` field contains the playhead location, in seconds, of the VideoDisplay control with which the cue point is associated.

When the playhead of the VideoDisplay control reaches a cue point, it dispatches a `cuePoint` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayCP.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.events.CuePointEvent;
            import mx.controls.videoClasses.CuePointManager;

            private function cpHandler(event:CuePointEvent):void {
                cp.text="got to cuepoint: " + event.cuePointName + " " +
                    String(event.cuePointTime);
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <mx:VideoDisplay
            source="http://localhost:8100/flex/assets/MyVideo.flv"
            cuePointManagerClass="mx.controls.videoClasses.CuePointManager"
            cuePoint="cpHandler(event);">
            <mx:cuePoints>
                <mx:Object name="first" time="10"/>
                <mx:Object name="second" time="20"/>
            </mx:cuePoints>
        </mx:VideoDisplay>
        <mx:TextArea id="cp"/>
    </mx:VBox>
</mx:Application>
```

In this example, the event listener writes a String to the TextArea control when the control reaches a cue point. The String contains the name and time of the cue point.

Adding a cue point by using the CuePointManager class

You can set cue points for the VideoDisplay control by using the `cuePointManager` property. This property is of type CuePointManager, where the CuePointManager class defines methods that you use to programmatically manipulate cue points, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayCPManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.events.CuePointEvent;

            [Bindable]
            private var myCuePoints:Array = [
            { name: "first", time: 10},
            { name: "second", time: 20} ];

            // Set cue points using methods of the CuePointManager class.
            private function initCP():void {
                myVid.cuePointManager.setCuePoints(myCuePoints);
            }

            private var currentCP:Object=new Object();

            private function cpHandler(event:CuePointEvent):void {
                cp.text="go to cuepoint: " + event.cuePointName + " " +
                    String(event.cuePointTime);
                // Remove cue point.
                currentCP.name=event.cuePointName;
                currentCP.time=event.cuePointTime;
                myVid.cuePointManager.removeCuePoint(currentCP);
                // Display the number of remaining cue points.
                cp.text=cp.text + "\n" + "Cue points remaining: " +
                    String(myVid.cuePointManager.getCuePoints().length);
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <mx:VideoDisplay id="myVid"
            initialize="initCP();"
            cuePointManagerClass="mx.controls.videoClasses.CuePointManager"
            source="http://localhost:8100/flex/assets/MyVideo.flv"
            cuePoint="cpHandler(event);"/>
        <mx:TextArea id="cp" width="200" />
    </mx:VBox>
```

```
</mx:Application>
```

Streaming video from a camera

You can use the `VideoDisplay.attachCamera()` method to configure the control to display a video stream from a camera, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayCamera.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Define a variable of type Camera.
            import flash.media.Camera;
            public var cam:Camera;

            public function initCamera():void {
                // Initialize the variable.
                cam = Camera.getCamera();
                myVid.attachCamera(cam)
            }
        ]]>
    </mx:Script>

    <mx:VideoDisplay id="myVid"
        width="320" height="240"
        creationComplete="initCamera();"/>
</mx:Application>
```

In this example, you create a Camera object in the event handler for the `creationComplete` event of the VideoDisplay control, then pass the Camera object as the argument to the `attachCamera()` method.

Using the VideoDisplay control with Flash Media Server 2

You can use the VideoDisplay control to import a media stream from Macromedia® Flash® Media Server 2 from Adobe, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\videodisplay\VideoDisplayFMS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox>
        <mx:Label text="RTMP FMS 2.0"/>
        <mx:VideoDisplay
            autoBandWidthDetection="false"
            source="rtmp://localhost/videodisplay/bike.flv"/>
    </mx:HBox>
</mx:Application>
```

In this example, you place the bike.flv file in the directory Flash Media Server 2\applications\videodisplay\streams_definst_.

Notice that you explicitly set the `autoBandWidthDetection` property to `false`, its default value. When the `autoBandWidthDetection` property is `true`, you must create the server-side file main.asc in the directory Flash Media Server 2\applications\videodisplay\scripts, which implements the following functions:

```
application.onConnect = function(p_client, p_autoSenseBW) {}
application.calculateClientBw = function(p_client) {}
Client.prototype.getStreamLength = function(p_streamName) {}
```

The following example shows an implementation of main.asc:

```
application.onConnect = function(p_client, p_autoSenseBW) {
  //Add security code here.

  this.acceptConnection(p_client);

  if (p_autoSenseBW)
    this.calculateClientBw(p_client);
  else
    p_client.call("onBWDone");
}

Client.prototype.getStreamLength = function(p_streamName) {
  return Stream.length(p_streamName);
}

application.calculateClientBw = function(p_client) {
  // Add code to set the clients BandWidth.
  // Use p_client.getStats() which returns bytes_in
  // and bytes_Out and check your bandWidth using
```

```
  // p_client.call("onBWCheck", result, p_client.payload).
  p_client.call("onBWDone");
}
```

For more information on main.asc, see the Flash Media Server 2 documentation.

Specifying the AMF version for Flash Media Server

When you read from or write to a NetConnection, NetStream, or SharedObject object as binary data, the `objectEncoding` property of the object indicates which Action Message Format (AMF) version to use: the ActionScript 3.0 format or the ActionScript 1.0 or ActionScript 2.0 format. The possible value of the `objectEncoding` property include the following:

AMF0 Objects are serialized using the AMF format for ActionScript 1.0 and 2.0.

AMF3 Objects are serialized using the AMF format for ActionScript 3.0.

DEFAULT Objects are serialized using the default format for your version of Flash Player.

All versions of Flash Media Server use the `AMF0` encoding. Therefore, Flex applications must set the `objectEncoding` property of all NetConnection and SharedObject objects used with Flash Media Server to `AMF0`. For NetStream, the `objectEncoding` property is read only. Any FAP connections or RTMP connections to your own server can use AMF3.

ColorPicker control

The ColorPicker control lets users select a color from a drop-down swatch panel (palette). It initially appears as a preview sample with the selected color. When a user selects the control, a color swatch panel appears. The panel includes a sample of the selected color and a color swatch panel. By default, the swatch panel displays the web-safe colors (216 colors, where each of the three primary colors has a value that is a multiple of 33, such as #CC0066)

For complete reference information, see ColorPicker in *Adobe Flex 2 Language Reference*.

About the ColorPicker control

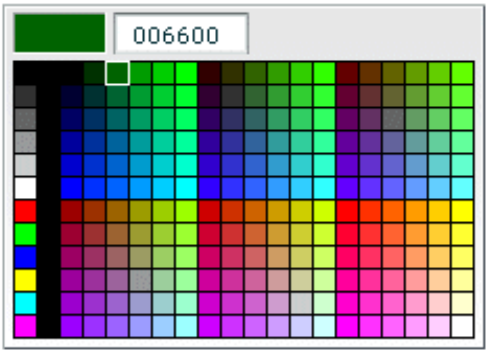
When you open the ColorPicker control, the swatch panel expands over other controls on the application, and normally opens downwards. If the swatch panel would hit the lower boundary of the application, but could fit above color picker button, it opens upward.

If you set the `showTextField` property to `true` (the default), the panel includes a text box with a label for the selected color. If you display a text box and set the `editable` property to `true` (the default), the user can specify a color by entering a hexadecimal value.

The following example shows a collapsed and expanded ColorPicker control that uses the default settings of the `mx:ColorPicker` tag:

Collapsed

Expanded



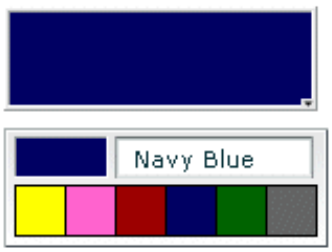
Flex populates the color swatch panel and the text box from a data provider. By default, the control uses a data provider that includes all the web-safe colors. If you use your own data provider you can specify the following:

The colors to display You *must* specify the colors if you use your own dataProvider.

Labels to display in the text box for the colors If you do not specify text labels, Flex uses the hexadecimal color values.

Additional information for each color This information can include any information that is of use to your application, such as IDs or descriptive comments.

The following image shows an expanded ColorPicker control that uses a custom data provider that includes color label values. It also uses styles to set the sizes of the display elements.



The ColorPicker control has the following default sizing characteristics:

Property	Default value
Default size	ColorPicker: 22 by 22 pixels Swatch panel: Sized to fit the ColorPicker control width
Minimum size	0 by 0
Maximum size	Undefined

Creating a ColorPicker control

You use the `<mx:ColorPicker>` tag to define a ColorPicker control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. For example, the ColorPicker control in the image was generated using the following, minimal, code:

```
<mx:ColorPicker id="cp"/>
```

The ColorPicker control uses a list-based data provider for the colors. For more information on this type of data provider, see Chapter 7, "Using Data Providers and Collections," on page 161. If you omit the data provider, the control uses a default data provider with the web-safe colors. The data provider can be an array of colors or an array of objects. The following example populates a ColorPicker with a simple array of colors. For information on using a more complex dataProvider, see "Using Objects to populate a ColorPicker control" on page 346.

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            [Bindable]
           public var simpleDP:Array = ['0x000000', '0xFF0000', '0xFF8800',
               '0xFFFF00', '0x88FF00', '0x00FF00', '0x00FF88', '0x00FFFF',
               '0x0088FF', '0x0000FF', '0x8800FF', '0xFF00FF', '0xFFFFFF'];
        ]]>
    </mx:Script>

    <mx:ColorPicker id="cp" dataProvider="{simpleDP}"/>
</mx:Application>
```

NOTE

You can also specify the data for the ColorPicker control using a <mx:dataProvider> child tag; for an example, see "Using custom field names" on page 348.
Some examples in this section use simple Arrays as custom data sources, not ArrayCollections, because the array contents are static.

You typically use events to handle user interaction with a ColorPicker control. The following example adds an event listener for a `change` event and an `open` event to the previous example ColorPicker control:

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPEvents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            //Import the event classes.
            import mx.events.DropdownEvent;
            import mx.events.ColorPickerEvent;

            [Bindable]
           public var simpleDP:Array = ['0x000000', '0xFF0000', '0xFF8800',
               '0xFFFF00', '0x88FF00', '0x00FF00', '0x00FF88', '0x00FFFF',
               '0x0088FF', '0x0000FF', '0x8800FF', '0xFF00FF', '0xFFFFFF'];

            public function openEvt(event:DropdownEvent):void {
                forChange.text="Opened";
            }

            public function changeEvt(event:ColorPickerEvent):void {
                forChange.text="Selected Item: "
                   + event.currentTarget.selectedItem + " Selected Index: "
                   + event.currentTarget.selectedIndex;
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <mx:TextArea id="forChange"
            width="150"/>
        <mx:ColorPicker id="cp"
            dataProvider="{simpleDP}"
            open="openEvt(event);"
            change="changeEvt(event);"/>
    </mx:VBox>
</mx:Application>
```

The ColorPicker control dispatches `open` event when the swatch panel opens and Dispatches a `change` event when the value of the control changes due to user interaction. The `currentTarget` property of the object passed to the event listener contains a reference to the ColorPicker control. In this example, the event listeners use two properties of the ColorPicker control, `selectedItem` and `selectedIndex`. Every `change` event updates the TextArea control with the selected item and the item's index in the control, and an `open` event displays the word *Opened*.

If you populate the ColorPicker control from an array of color values, the `target.selectedItem` field contains the hexadecimal color value. If you populate it from an array of Objects, the `target.selectedItem` field contains a reference to the object that corresponds to the selected item.

The index of items in the ColorPicker control is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items; therefore, the `target.selectedIndex` value is zero-based, and a value of 2 in the preceding example refers to the data provider entry with color 0xFF8800.

Using Objects to populate a ColorPicker control

You can populate a ColorPicker control with an Array of Objects. By default, the ColorPicker uses two fields in the Objects: one named `color`, and another named `label`. The `label` field value determines the text in the swatch panel's text field. If the Objects do not have a `label` field, the control uses the `color` field value in the text field. You can use the ColorPicker control's `colorField` and `labelField` properties to specify different names for the color and label fields. The Objects can have additional fields, such as a color description or an internal color ID, that you can use in your ActionScript.

Example: ColorPicker control that uses Objects

The following example shows a ColorPicker that uses an Array of Objects with three fields: `color`, `label`, and `descript`.

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.events.ColorPickerEvent;
            import mx.events.DropdownEvent;

            [Bindable]
            public var complexDPArray:Array = [
                {label:"Yellow", color:"0xFFFF00",
                    descript:"A bright, light color."},
                {label:"Hot Pink", color:"0xFF66CC",
                    descript:"It's HOT!"},
                {label:"Brick Red", color:"0x990000",
                    descript:"Goes well with warm colors."},
                {label:"Navy Blue", color:"0x000066",
                    descript:"The conservative favorite."},
                {label:"Forest Green", color:"0x006600",
                    descript:"Great outdoorsy look."},
```

```
                {label:"Grey", color:"0x666666",
                    descript:"An old reliable."}]

            public function openEvt(event:DropdownEvent):void {
                descriptBox.text="";
            }

            public function changeEvt(event:ColorPickerEvent):void {
                descriptBox.text=event.currentTarget.selectedItem.label
                    + ": " + event.currentTarget.selectedItem.descript;
            }
        ]]>
    </mx:Script>

    <!-- Convert the Array to an ArrayCollection. Do this if
    you might change the colors in the panel dynamically. -->
    <mx:ArrayCollection id="complexDP" source="{complexDPArray}"/>

    <mx:VBox>
        <mx:TextArea id="descriptBox"
            width="150" height="50"/>
        <mx:ColorPicker id="cp"
            height="50" width="150"
            dataProvider="{complexDP}"
            change="changeEvt(event);"
            open="openEvt(event);"
            swatchWidth="25"
            swatchHeight="25"
            textFieldWidth="95"
            editable="false"/>
    </mx:VBox>
</mx:Application>
```

In this example, the `selectedItem` property contains a reference to the object defining the selected item. The example uses `selectedItem.label` to access the object's `label` property (the color name), and `selectedItem.descript` to access the object's `descript` property (the color description). Every `change` event updates the `TextArea` control with the `label` property of the selected item and the item's description. The `open` event clears the current text in the `TextArea` control each time the user opens up the ColorPicker to display the swatch panel.

This example also uses several of the ColorPicker properties and styles to specify the control's behavior and appearance. The `editable` property prevents users from entering a value in the color label box (so they can only select the colors from the dataProvider). The `swatchWidth` and `swatchHeight` styles control the size of the color samples in the swatch panel, and the `textFieldWidth` style ensures that the text field is long enough to accommodate the longest color name.

Using custom field names

In some cases, you might want to use custom names for the color and label fields; for example, if the data comes from an external data source with custom column names. The following code changes the previous example to use custom color and label fields called cName and cVal. It also shows how to use an `<mx:dataProvider>` tag to populate the data provider.

```
<?xml version="1.0"?>
<!-- controls\colorpicker\CPCustomFieldNames.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.events.ColorPickerEvent;
            import mx.events.DropdownEvent;

            public function openEvt(event:DropdownEvent):void {
                descriptBox.text="";
            }

            public function changeEvt(event:ColorPickerEvent):void {
                descriptBox.text=event.currentTarget.selectedItem.cName
                    + ": " + event.currentTarget.selectedItem.cDescript;
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <mx:TextArea id="descriptBox"
            width="150" height="50"/>
        <mx:ColorPicker id="cp"
            height="50" width="150"
            labelField="cName"
            colorField="cVal"
            change="changeEvt(event)"
            open="openEvt(event)"
            swatchWidth="25"
            swatchHeight="25"
            textFieldWidth="95"
            editable="false">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object cName="Yellow" cVal="0xFFFF00"
                            cDescript="A bright, light color."/>
                        <mx:Object cName="Hot Pink" cVal="0xFF66CC"
                            cDescript="It's HOT!"/>
                        <mx:Object cName="Brick Red" cVal="0x990000"
                            cDescript="Goes well with warm colors."/>
                        <mx:Object cName="Navy Blue" cVal="0x000066"
```

```
                    cDescript="The conservative favorite."/>
                <mx:Object cName="Forest Green" cVal="0x006600"
                    cDescript="Great outdoorsy look."/>
                <mx:Object cName="Grey" cVal="0x666666"
                    cDescript="An old reliable."/>
            </mx:source>
        </mx:ArrayCollection>
    </mx:dataProvider>
</mx:ColorPicker>
</mx:VBox>
</mx:Application>
```

User interaction

A ColorPicker control can be editable or noneditable. In a noneditable ColorPicker control, the user must select a color from among the swatch panel options. In an editable ColorPicker control, a user can select swatch panel items or enter a hexadecimal color value directly into the label text field at the top of the swatch panel. Users can type numbers and uppercase or lowercase letters in the ranges a-f and A-F in the text box; it ignores all other non-numeric characters.

Mouse interaction

You can use the mouse to navigate and select from the control:

- Click the collapsed control to display or hide the swatch panel.
- Click any swatch in the swatch panel to select it and close the panel.
- Click outside the panel area to close the panel without making a selection.
- Click in the text field to move the text entry cursor.

Keyboard interaction

If the ColorPicker is editable, and the swatch panel has the focus, alphabetic keys in the range A-F and a-f, numeric keys, and the Backspace and Delete keys enter and remove text in the color text box. You can also use the following keystrokes to control the ColorPicker:

Key	Description
Control+Down Arrow	Opens the swatch panel and puts the focus on the selected swatch.
Control+Up Arrow	Closes the swatch panel, if open.
Home	Moves the selection to the first color in a row of the swatch panel. Has no effect if there is a single column.

Key	Description
End	Moves the selection to the last color in a row of the swatch panel. Has no effect if there is a single column.
Page Up	Moves the selection to the top color in a column of the swatch panel. Has no effect if there is a single row.
Page Down	Moves the selection to the bottom color in a column of the swatch panel. Has no effect if there is a single row.
Escape	Closes the swatch panel without changing the color in the color picker. Most Web browsers do not support using his key.
Enter	Selects the current color from the swatch panel and closes the swatch panel; equivalent to clicking a color swatch. If the focus is on the text field of an editable ColorPicker, selects the color specified by the field text.
Arrows	When the swatch panel is open, moves the focus to the next color left, right, up, and down in the swatch grid. On a single-row swatch panel, Up and Right Arrow keys are equivalent, and Down and Left Arrow keys are equivalent. On a multirow swatch panel, the selection wraps to the beginning or end of the next or previous line. On a single-row swatch panel, pressing the key past the beginning or end of the row loops around on the row. When the swatch panel is closed, but has the focus, the Up and Down Arrow keys have no effect. The Left and Right Arrow keys change the color picker selection, moving through the colors as if the panel were open.

NOTE

When the swatch panel is open, you cannot use the Tab and Shift+Tab keys to move the focus to another object.

Alert control

All Flex components can call the static `show()` method of the Alert class to open a pop-up modal dialog box with a message and an optional title, buttons, and icons. The following example shows an Alert control pop-up dialog box:

The Alert control closes when you select a button in the control, or press the Escape key.

The `Alert.show()` method has the following syntax:

```
public static show(
  text:String,
  title:String = null,
  flags:uint = mx.controls.Alert.OK,
  parent:Sprite = null,
  clickListener:Function = null,
  iconClass:Class = null,
  defaultButton:uint = mx.controls.Alert.OK) : Alert
```

This method returns an Alert control object.

The following table describes the arguments of the `show()` method:

Argument	Description
`text`	(Required) Specifies the text message displayed in the dialog box.
`title`	Specifies the dialog box title. If omitted, displays a blank title bar.
`flags`	Specifies the button(s) to display in the dialog box. The options are as follows: `mx.controls.Alert.OK` OK button `mx.controls.Alert.YES` Yes button `mx.controls.Alert.NO` No button `mx.controls.Alert.CANCEL` Cancel button Each option is a bit value and can be combined with other options using the pipe '|' operator. The buttons will appear in the order listed here regardless of the order you specify them in your code.The default value is `mx.controls.Alert.OK`.
`parent`	The parent object of the Alert control.

Argument	Description
clickListener	Specifies the listener for click events from the buttons. The event object passed to this handler is an instance of the CloseEvent class. The event object contains the detail field, which is set to the button flag that was clicked (mx.controls.Alert.OK, mx.controls.Alert.CANCEL, mx.controls.Alert.YES, or mx.controls.Alert.NO).
iconClass	Specifies an icon to display to the left of the message text in the dialog box.
defaultButton	Specifies the default button using one of the valid values for the flags argument. This is the button that is selected when the user presses the Enter key. The default value is Alert.OK. Pressing the Escape key triggers the Cancel or No button just as if you clicked it.

To use the Alert control, you first import the Alert class into your application. then call the show() method, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:TextInput id="myInput"
        width="150"
        text=""/>
    <mx:Button id="myButton"
        label="Copy Text"
        click="myText.text = myInput.text;
          Alert.show('Text Copied!', 'Alert Box', mx.controls.Alert.OK);"/>
    <mx:TextInput id="myText"/>
</mx:Application>
```

In this example, selecting the Button control copies text from the TextInput control to the TextArea control, and displays the Alert control.

You can also define an event listener for the Button control, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertSimpleEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;

            private function alertListener():void {
                myText.text = myInput.text;
                Alert.show("Text Copied!", "Alert Box", Alert.OK);
            }
        ]]>
    </mx:Script>

    <mx:TextInput id="myInput"
        width="150"
        text=""/>
    <mx:Button id="myButton"
        label="Copy Text"
        click="alertListener();"/>
    <mx:TextInput id="myText"/>
</mx:Application>
```

NOTE After the `show()` method creates the dialog box, Flex continues processing of your application; it does not wait for the user to close the dialog box.

Sizing the Alert control

The Alert control automatically sizes itself to fit its text, buttons, and icon. You can explicitly size an Alert control using the Alert object returned from the `show()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;

            // Define variable to hold the Alert object.
            public var myAlert:Alert;

            private function openAlert():void {
                myAlert = Alert.show("Copy Text?", "Alert",
                    Alert.OK    | Alert.CANCEL);
                // Set the height and width of the Alert control.
                myAlert.height=150;
                myAlert.width=150;
            }
        ]]>
    </mx:Script>

        <mx:TextInput id="myInput"
            width="150"
            text=""/>
        <mx:Button id="myButton"
            label="Copy Text"
            click="openAlert();"/>
        <mx:TextInput id="myText"/>
</mx:Application>
```

In this example, you set the `height` and `width` properties of the Alert object to explicitly size the control.

Using event listeners with the Alert control

The next example adds an event listener to the Alert control pop-up dialog box. An event listener lets you perform processing when the user selects a button of the Alert control. The event object passed to the event listener is of type CloseEvent.

In the next example, you only copy the text when the user selects the OK button in the Alert control:

```
<?xml version="1.0"?>
<!-- controls\alert\AlertEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.CloseEvent;

            private function alertListener(eventObj:CloseEvent):void {
                // Check to see if the OK button was pressed.
                if (eventObj.detail==Alert.OK) {
                    myText.text = myInput.text;
                }
            }
        ]]>
    </mx:Script>

    <mx:TextInput id="myInput"
        width="150"
        text="" />
    <mx:Button id="myButton"
        label="Copy Text"
        click='Alert.show("Copy Text?", "Alert",
            Alert.OK | Alert.CANCEL, this,
            alertListener, null, Alert.OK);'/>
    <mx:TextInput id="myText"/>
</mx:Application>
```

In this example, you define an event listener for the Alert control. Within the body of the event listener, you determine which button was pressed by examining the `detail` property of the event object. The event object is an instance of the CloseEvent class. If the user pressed the OK button, copy the text. If the user pressed any other button, or pressed the Escape key, do not copy the text.

Specifying an Alert control icon

You can include an icon in the Alert control that appears to the left of the Alert control text. This example modifies the example from the previous section to add the `Embed` metadata tag to import the icon. For more information on importing resources, see Chapter 4, "Using ActionScript," on page 55.

```
<?xml version="1.0"?>
<!-- controls\alert\AlertIcon.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.CloseEvent;

            [Embed(source="assets/alertIcon.jpg")]
            [Bindable]
            public var iconSymbol:Class;

            private function alertListener(eventObj:CloseEvent):void {
                // Check to see if the OK button was pressed.
                if (eventObj.detail==Alert.OK) {
                    myText.text = myInput.text;
                }
            }
        ]]>
    </mx:Script>

        <mx:TextInput id="myInput"
            width="150"
            text=""/>
        <mx:Button id="myButton"
            label="Copy Text"
            click='Alert.show("Copy Text?", "Alert",
                Alert.OK | Alert.CANCEL, this,
                alertListener, iconSymbol,  Alert.OK );'/>
        <mx:TextInput id="myText"/>
</mx:Application>
```

ProgressBar control

The ProgressBar control provides a visual representation of the progress of a task over time. There are two types of ProgressBar controls: determinate and indeterminate. A *determinate* ProgressBar control is a linear representation of the progress of a task over time. You can use this when the user is required to wait for an extended period of time, and the scope of the task is known.

An *indeterminate* ProgressBar control represents time-based processes for which the scope is not yet known. As soon as you can determine the scope, you should use a determinate ProgressBar control.

The following example shows both types of ProgressBar controls:

Determinate ProgressBar control

Indeterminate ProgressBar control

Use the ProgressBar control when the user is required to wait for completion of a process over an extended period of time. You can attach the ProgressBar control to any kind of loading content. A label can display the extent of loaded contents when enabled.

The ProgressBar control has the following default properties:

Property	Default value
default size	150 pixels wide and 4 pixels high
Minimum size	0
Maximum size	Undefined

ProgressBar control modes

You use the `mode` property to specify the operating mode of the ProgressBar control. The ProgressBar control supports the following modes of operation:

event Use the `source` property to specify a loading process that emits `progress` and `complete` events. For example, the SWFLoader and Image controls emit these events as part of loading a file. You typically use a determinate ProgressBar in this mode. This is the default mode.

You also use this mode if you want to measure progress on multiple loads; for example, if you reload an image, or use the SWFLoader and Image controls to load multiple images.

polled Use the `source` property to specify a loading process that exposes the `getBytesLoaded()` and `getsBytesTotal()` methods. For example, the SWFLoader and Image controls expose these methods. You typically use a determinate ProgressBar in this mode.

manual Set the `maximum`, `minimum`, and `indeterminate` properties along with calls to the `setProgress()` method. You typically use an indeterminate ProgressBar in this mode.

Creating a ProgressBar control

You use the `<mx:ProgressBar>` tag to define a ProgressBar control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example uses the default event mode to track the progress of loading an image using the Image control:

```
<?xml version="1.0"?>
<!-- controls\pbar\PBarSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function initImage():void {
             image1.load('http://localhost:8100/flex/assets/DSC00034.JPG');
            }
        ]]>
    </mx:Script>

    <mx:VBox id="vbox0"
        width="600" height="600">
        <mx:Canvas>
            <mx:ProgressBar width="200" source="image1"/>
        </mx:Canvas>
        <mx:Button id="myButton"
            label="Show"
            click="initImage();"/>
        <mx:Image id="image1"
            height="600" width="600"
            autoLoad="false"
            visible="true"/>
    </mx:VBox>
</mx:Application>
```

In this mode, the Image control issues `progress` events during the load, and a `complete` event when the load completes.

The `<mx:Image>` tag exposes the `getBytesLoaded()` and `getBytesTotal()` methods, so you could also use `polled` mode, as the following example shows:

```
<mx:ProgressBar width="200" source="image1" mode="polled"/>
```

In manual mode, `mode="manual"`, you use an indeterminate ProgressBar control with the `maximum` and `minimum` properties and the `setProgress()` method. The `setProgress()` method has the following method signature:

```
setProgress(Number completed, Number total)
```

completed Specifies the progress made in the task, and must be between the `maximum` and `minimum` values. For example, if you were tracking the number of bytes to load, this would be the number of bytes already loaded.

total Specifies the total task. For example, if you were tracking bytes loaded, this would be the total number of bytes to load. Typically, this is the same value as `maximum`.

To measure progress, you make explicit calls to the `setProgress()` method to update the ProgressBar control.

Defining the label of a ProgressBar control

By default, the ProgressBar displays the label *LOADING xx%*, where *xx* is the percent of the image loaded. You use the `label` property to specify a different text string to display.

The `label` property lets you include the following special characters in the label text string:

%1 Corresponds to the current number of bytes loaded.

%2 Corresponds to the total number of bytes.

%3 Corresponds to the percent loaded.

%% Corresponds to the % sign.

For example, to define a label that displays as

```
Loading Image 1500 out of 78000 bytes, 2%
```

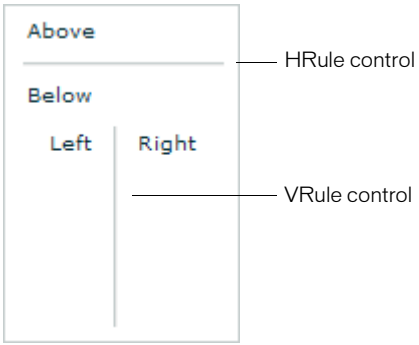
use the following code:

```
<?xml version="1.0"?>
<!-- controls\pbar\PBarLabel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function initImage():void {
             image1.load('http://localhost:8100/flex/assets/DSC00034.JPG');
            }
        ]]>
    </mx:Script>

    <mx:VBox id="vbox0"
        width="600" height="600">
        <mx:Canvas>
            <mx:ProgressBar
                width="300"
                source="image1"
                mode="polled"
                label="Loading Image %1 out of %2 bytes, %3%%"
                labelWidth="400"/>
        </mx:Canvas>
        <mx:Button id="myButton"
            label="Show"
            click="initImage();"/>
        <mx:Image id="image1"
            height="600" width="600"
            autoLoad="false"
            visible="true"/>
    </mx:VBox>
</mx:Application>
```

HRule and VRule controls

The HRule (Horizontal Rule) control creates a single horizontal line and the VRule (Vertical Rule) control creates a single vertical line. You typically use these controls to create dividing lines within a container.

The following image shows an HRule and a VRule control:



The HRule and VRule controls have the following default properties:

Property	Default value
Default size	**Horizontal Rule** The default width is 100 pixels, and the default height is 2 pixels. By default, the HRule control is not resizable; set `width` and `height` to percentage values to enable resizing. **Vertical Rule** The default height is 100 pixels, and the default width is 2 pixels. By default, the VRule control is not resizable; set `width` and `height` to percentage values to enable resizing.
`strokeWidth`	2 pixels
`strokeColor`	`0xC4CCCC`
`shadowColor`	`0xEEEEEE`

Creating HRule and VRule controls

You define HRule and VRule controls in MXML using the `<mx:HRule>` and `<mx:VRule>` tags, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

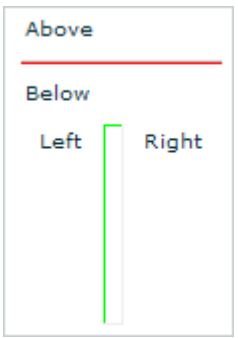
```
<?xml version="1.0"?>
<!-- controls\rule\RuleSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <mx:Label text="Above"/>
        <mx:HRule/>
        <mx:Label text="Below"/>
    </mx:VBox>
    <mx:HBox>
        <mx:Label text="Left"/>
        <mx:VRule/>
        <mx:Label text="Right"/>
    </mx:HBox>
</mx:Application>
```

This example creates the output shown in the preceding image.

You can also use properties of the HRule and VRule controls to specify line width, stroke color, and shadow color, as the following example shows:

```
<?xml version="1.0"?>
<!-- controls\rule\RuleProps.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <mx:Label text="Above"/>
        <mx:HRule shadowColor="0xEEEEEE"/>
        <mx:Label text="Below"/>
    </mx:VBox>
    <mx:HBox>
        <mx:Label text="Left"/>
        <mx:VRule strokeWidth="10" strokeColor="0xC4CCCC"/>
        <mx:Label text="Right"/>
    </mx:HBox>
</mx:Application>
```

This code produces the following image:

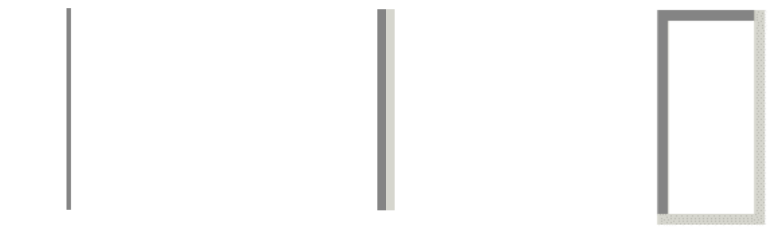


Sizing HRule and VRule controls

For the HRule and VRule controls, the `strokeWidth` property determines how Flex draws the line, as follows:

- If you set the `strokeWidth` property to 1, Flex draws a 1-pixel-wide line.
- If you set the `strokeWidth` property to 2, Flex draws the rule as two adjacent 1-pixel-wide lines, horizontal for an HRule control or vertical for a VRule control. This is the default value.
- If you set the `strokeWidth` property to a value greater than 2, Flex draws the rule as a hollow rectangle with 1-pixel-wide edges.

The following example shows all three options:



If you set the `height` property of an HRule control to a value greater than the `strokeWidth` property, Flex draws the rule within a rectangle of the specified height, and centers the rule vertically within the rectangle. The height of the rule is the height specified by the `strokeWidth` property.

If you set the `width` property of a VRule control to a value greater than the `strokeWidth` property, Flex draws the rule within a rectangle of the specified width, and centers the rule horizontally within the rectangle. The width of the rule is the width specified by the `strokeWidth` property.

If you set the `height` property of an HRule control or the `width` property of a VRule control to a value smaller than the `strokeWidth` property, the rule is drawn as if it had a `strokeWidth` property equal to the `height` or `width` property.

NOTE

If the `height` and `width` properties are specified as percentage values, the actual pixel values are calculated before the `height` and `width` properties are compared to the `strokeWidth` property.

The `strokeColor` and `shadowColor` properties determine the colors of the HRule and VRule controls. The `strokeColor` property specifies the color of the line as follows:

- If you set the `strokeWidth` property to 1, specifies the color of the entire line.
- If you set the `strokeWidth` property to 2, specifies the color of the top line for an HRule control, or the left line for a VRule control.
- If you set the `strokeWidth` property to a value greater than 2, specifies the color of the top and left edges of the rectangle.

The `shadowColor` property specifies the shadow color of the line as follows:

- If you set the `strokeWidth` property to 1, does nothing.
- If you set the `strokeWidth` property to 2, specifies the color of the bottom line for an HRule control, or the right line for a VRule control.
- If you set the `strokeWidth` property to a value greater than 2, specifies the color of the bottom and right edges of the rectangle.

Setting style properties

The `strokeWidth`, `strokeColor`, and `shadowColor` properties are style properties. Therefore, you can set them in MXML as part of the tag definition, set them using the `<mx:Style>` tag in MXML, or set them using the `setStyle()` method in ActionScript.

The following example uses the `<mx:Style>` tag to set the default value of the `strokeColor` property of all HRule controls to `#00FF00` (lime green), and the default value of the `shadowColor` property to `#0000FF` (blue). This example also defines a class selector, called `thickRule`, with a `strokeWidth` of 5 that you can use with any instance of an HRule control or VRule control:

```
<?xml version="1.0"?>
<!-- controls\rule\RuleStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        .thickRule {strokeWidth:5}
        HRule {strokeColor:#00FF00; shadowColor:#0000FF}
    </mx:Style>

    <mx:HRule styleName="thickRule"/>
</mx:Application>
```

This example produces the following image:

ScrollBar control

The VScrollBar (vertical ScrollBar) control and HScrollBar (horizontal ScrollBar) controls let the user control the portion of data that is displayed when there is too much data to fit in the display area.

Although you can use the VScrollBar control and HScrollBar control as stand-alone controls, they are usually combined with other components as part of a custom component to provide scrolling functionality. For more information, see *Creating and Extending Flex 2 Components*.

ScrollBar controls consists of four parts: two arrow buttons, a track, and a thumb. The position of the thumb and display of the buttons depends on the current state of the ScrollBar control. The ScrollBar control uses four parameters to calculate its display state:

- Minimum range value
- Maximum range value
- Current position; must be within the minimum and maximum range values
- Viewport size; represents the number of items in the range that can be displayed at once and must be equal to or less than the range

Creating a ScrollBar control

You define a ScrollBar control in MXML using the `<mx:VScrollbar>` tag for a vertical ScrollBar or the `<mx:HScrollBar>` tag for a horizontal ScrollBar, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- controls\bar\SBarSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

     <mx:Script>
        <![CDATA[

            import mx.events.ScrollEvent;

            // Event handler function to display the scroll location.
            private function myScroll(event:ScrollEvent):void {
               showPosition.text = "VScrollBar properties summary:" + '\n' +
                     "-------------------------------------" + '\n' +
                     "Current scroll position: " +
                     event.currentTarget.scrollPosition  + '\n' +
                     "The maximum scroll position: " +
                     event.currentTarget.maxScrollPosition + '\n' +
                     "The minimum scroll position: " +
                     event.currentTarget.minScrollPosition;
            }
        ]]>
    </mx:Script>

    <mx:Label
        width="100%"
        color="blue"
        text="Click on the scroll bar to view its properties."/>

    <mx:VScrollBar id="bar"
        height="100%"
        minScrollPosition="0"
        maxScrollPosition="{this.width - 20}"
        lineScrollSize="50"
        pageScrollSize="100"
        repeatDelay="1000"
        repeatInterval="500"
        scroll="myScroll(event);"/>

    <mx:TextArea id="showPosition"
        height="100%" width="100%"
        color="blue"/>
</mx:Application>
```

Sizing a ScrollBar control

The ScrollBar control does not display correctly if it is sized smaller than the height of the up arrow and down arrow buttons. There is no error checking for this condition. Adobe recommends that you hide the ScrollBar control in such a condition. If there is not enough room for the thumb, the thumb is made invisible.

User interaction

Use the mouse to click the various portions of the ScrollBar control, which dispatches events to listeners. The object listening to the ScrollBar control is responsible for updating the portion of data displayed. The ScrollBar control updates itself to represent the new state after the action has taken place.

CHAPTER 10

Using Text Controls

10

Text controls can display text, let the user enter text, or do both. This topic describes how to use text controls in an Adobe Flex application.

Contents

About text controls

You use Flex text-based controls to display text and to let users enter text into your application. The following table lists the controls, and indicates whether the control can have multiple lines of input instead of a single line of text, and whether the control can accept user input:

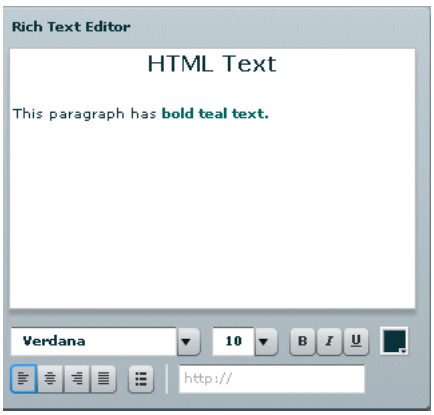
Control	Multiline	Allows user Input
Label	No	No
TextInput	No	Yes
Text	Yes	No
TextArea	Yes	Yes
RichTextEditor	Yes	Yes

All controls except the RichTextEditor control are single components with a simple text region; for example, the following image shows a TextInput control in a simple form:

The following code produces the preceding image:

```
<?xml version="1.0"?>
<!-- textcontrols/FormItemLabel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
  <mx:Form id="myForm" width="500" backgroundColor="#909090">
     <!-- Use a FormItem to label the field. -->
     <mx:FormItem label="First Name">
        <mx:TextInput id="ti1" width="150"/>
     </mx:FormItem>
  </mx:Form>
</mx:Application>
```

The RichTextEditor control is a compound control; it consists of a Panel control that contains a TextArea control and a ControlBar with several controls for specifying the text format and HTTP links. The following image shows a RichTextEditor control:



The following code produces the preceding image:

```
<?xml version="1.0"?>
<!-- textcontrols/RTECDATA.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
  <mx:RichTextEditor id="rte1" title="Rich Text Editor">
     <mx:htmlText>
        <![CDATA[
           <p align='center'><b><font size='16'>HTML Text</font></b>
           </p>This paragraph has <font color='#006666'><b>bold teal text.
           </b></font>
        ]]>
     </mx:htmlText>
  </mx:RichTextEditor>
</mx:Application>
```

Flex text-based controls let you set and get text by using the following properties:

text Plain text without formatting information. For information on using the `text` property, see "Using the text property" on page 371.

htmlText Rich text that represents formatting by using a subset of HTML tags, and can include bulleted text and URL links. For information on using the `htmlText` property, see "Using the htmlText property" on page 376.

Both properties set the same underlying text, but you can use different formats. For example, you can do the following to set, modify, and get text:

- You can set formatted text by using the `htmlText` property, and get it back as a plain text string by using the `text` property.
- You can set formatted text in user-editable text controls (TextInput, TextArea, RichTextEditor) by setting the text string with the `text` property and formatting a section of this text by using the TextRange class. If you get the text back by using the `htmlText` property, the property string includes HTML tags for the formatting. For more information on using the TextRange class, see "Selecting and modifying text" on page 387.

Using the text property

You can use the `text` property to specify the text string that appears in a text control or to get the text in the control as a plain text String. When you set this property, any HTML tags in the text string appear in the control as literal text.

You cannot specify text formatting when you set the `text` property, but you can format the text in the control. You can use the text control styles to format all of the text in the control, and you can use the TextRange class to format ranges of text. (For more information on using the TextRange class, see "Selecting and modifying text" on page 387.)

The following code line uses a `text` property to specify label text:

```
<mx:Label text="This is a simple text label"/>
```

The way you specify special characters, including quotation marks, greater than and less than signs, and apostrophes, depends on whether you use them in MXML tags or in ActionScript. It also depends on whether you specify the text directly or wrap the text in a CDATA section.

NOTE If you specify the value of the `text` property by using a string directly in MXML, Flex collapses white space characters. If you specify the value of the `text` property in ActionScript, Flex does not collapse white space characters.

Specifying special characters in the text property

The following rules specifying how to include special characters in the `text` property of a text control MXML tag, either in a property assignment, such as `text="the text"`, or in the body of an `<mx:text>` subtag.

In standard text The following rules determine how you use special characters if you do not use a CDATA section.

- To use the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), insert the XML character entity equivalents of `<`, `>`, and `&`, respectively. You can also use `&quot;` and `&apos;` for double-quotation marks (") and single-quotation marks ('), and you can use numeric character references, such as &#165 for the Yen mark (¥). Do not use any other named character entities; Flex treats them as literal text.
- You cannot use the character that encloses the property text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence `\"` for any double-quotation marks in the string. If you surround the string in single-quotation marks (') use the escape sequence `\'` for any single-quotation marks in the string. You *can* use single-quotation marks inside a string that is surrounded in double-quotation marks, and double-quotation marks inside a string that is surrounded in single-quotation marks.
- Flex text controls ignore escape characters such as \t or \n in the `text` property. They ignore or converts to spaces, tabs and line breaks, depending on whether you are specifying a property assignment or an `<mx:text>` subtag. To include line breaks, put the text in a CDATA section. In the Text control `text="`*`string`*`"` attribute specifications, you can also specify them as numeric character entities, such as
 for a Return character or 	 for a Tab character, but you cannot do this in an `<mx:text>` subtag.

The following code example uses the `text` property with standard text:

```
<?xml version="1.0"?>
<!-- textcontrols/StandardText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="400">

  <mx:Text width="400" text="This string contains a less than, &lt;,
     greater than, &gt;, ampersand, &amp;, apostrophe, ', and
     quotation mark &quot;."/>

  <mx:Text width="400" text='This string contains a less than, &lt;,
     greater than, &gt;, ampersand, &amp;, apostrophe, &apos;, and
     quotation mark, ".'/>

  <mx:Text width="400">
     <mx:text>
        This string contains a less than, &lt;, greater than,
        &gt;, ampersand, &amp;, apostrophe, ', and quotation mark, ".
     </mx:text>
  </mx:Text>

</mx:Application>
```

The resulting application contains three almost identical text controls, each with the following text. The first two controls, however, convert any tabs in the text to spaces.

```
This string contains a less than, <, greater than, >, ampersand, &,
  apostrophe, ', and  quotation mark, ".
```

In a CDATA section If you wrap the text string in the `CDATA` tag, the following rules apply:

- You cannot use a CDATA section in a property assignment statement in the text control opening tag; you must define the property in an `<mx:text>` child tag.
- Text inside the CDATA section appears as it is entered, including white space characters. Use literal characters, such as " or < for special characters, and use standard return and tab characters. Character entities, such as >, and backslash-style escape characters, such as \n, appear as literal text.

The following code example follows these CDATA section rules. The second and third lines of text in the <mx:text> tag are not indented because any leading tab or space characters would appear in the displayed text.

```
<?xml version="1.0"?>
<!-- textcontrols/TextCDATA.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500">
  <mx:Text width="100%">
     <mx:text>
        <![CDATA[This string contains a less than, <, greater than, >,
ampersand, &, apostrophe, ', return,
tab. and quotation mark, ".]]>
     </mx:text>
  </mx:Text>
</mx:Application>
```

The displayed text appears on three lines, as follows:

```
This string contains a less than, <, greater than, >,
ampersand, &, apostrophe, ', return,
tab.  and quotation mark, ".
```

Specifying special characters in ActionScript

The following rules specifying how to include special characters in a text control when you specify the control's `text` property value in ActionScript, for example, in an initialization function, or when assigning a string value to a variable that you use to populate the property.

- You cannot use the character that encloses the text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence \" for any double-quotation marks in the string. If you surround the string in single-quotation marks ('), use the escape sequence \' for any single-quotation marks in the string.
- Use backslash escape characters for special characters, including \t for the tab character, and \n or \r for a return/line feed character combination. You can use the escape character \" for the double-quotation mark and \' for the single-quotation mark.
- In standard text, but not in CDATA sections, you can use the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), by inserting the XML character entity equivalents of `<`, `>`, and `&`, respectively. You can also use `&quot;` and `&apos;` for double-quotation marks ("), and single-quotation marks ('), and you can use numeric character references, such as `&#165;` for the Yen mark (¥). Do not use any other named character entities; Flex treats them as literal text.
- In CDATA sections only, do not use character entities or references, such as `<` or ¥ because Flex treats them as literal text. Instead, use the actual character, such as <.

The following example uses an initialization function to set the `text` property to a string that contains these characters:

```
<?xml version="1.0"?>
<!-- textcontrols/InitText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initText()">
  <mx:Script>
     public function initText():void {
        //The following is on one line.
        myText.text="This string contains a return, \n, tab, \t, and
quotation mark, \". This string also contains less than, &lt;, greater than,
&gt;, ampersand, &amp;, and apostrophe, ', characters.";
     }
  </mx:Script>
  <mx:Text width="450" id="myText" initialize="initText();"/>
</mx:Application>
```

The following example uses an `<mx:Script>` tag with a variable in a CDATA section to set the `text` property:

```
<?xml version="1.0"?>
<!-- textcontrols/VarText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
  <![CDATA[
     [Bindable]
        //The following is on one line.
     public var myText:String ="This string contains a return, \n, tab, \t,
and quotation mark, \". This string also contains less than, <, greater
than, <, ampersand, <;, and apostrophe, ', characters.";
  ]]>
  </mx:Script>
  <mx:Text width="450" text="{myText}"/>
</mx:Application>
```

The displayed text for each example appears on three lines. The first line ends at the return specified by the \n character. The remaining text wraps onto a third line because it is too long to fit on a single line. (Note: Although the tab character may be noticeable in the following output, it is included in the right location.)

```
This string contains a return,
, tab, , and quotation mark, ". This string also contains less than, <,
greater than, >, ampersand, &, and apostrophe, ', characters.
```

Using the htmlText property

You use the `htmlText` property to set or get an HTML-formatted text string. You can also use one tag that is not part of standard HTML, the `textFormat` tag. For details of supported tags and attributes, see "Using tags in HTML text" on page 380.

You can also specify text formatting using Flex styles. You can set a base style, such as the font characteristics or the text weight using a style, and override the base style in sections of your text by using tags, such as the `<font>` tag. In the following example, the `<mx:Text>` tag styles specify blue, italic, 14 point text, and the `<mx:htmlText>` tag includes HTML tags that override the color and point size.

```
<?xml version="1.0"?>
<!-- textcontrols/HTMLTags.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="450"
backgroundGradientColors="[#FFFFFF, #FFFFFF]">
  <mx:Text width="100%" color="blue" fontStyle="italic" fontSize="14">
     <mx:htmlText>
        <![CDATA[This is 14 point blue italic text.<br><b><font
color="#000000" size="10">This text is 10 point black, italic, and bold.</
font></b>]]>
     </mx:htmlText>
  </mx:Text>
</mx:Application>
```

This code results in the following output:

Specifying HTML tags and text

To prevent the Flex compiler from generating errors when it encounters HTML tags in the text, use one of the following techniques:

- Wrap your text in a `CDATA` tag.
- Specify HTML markup by using the `&lt;`, `&gt;`, and `&amp;` character entities in place of the left angle bracket (<), right angle bracket (>), and ampersand (&) HTML delimiters.

Adobe recommends using CDATA sections for all but simple HTML markup, because the character entity technique has significant limitations:

- Extensive HTML markup can be cumbersome to write and difficult to read.
- You must use a complex escape sequence to include the less than and ampersand characters in your text.

For example, to display the following string:

A less than character < and **bold text**.

without using a CDATA section, you must use the following text:

```
A less than character &amp;c#060; and &lt;b&gtbold text&lt;/b&gt.
```

In a CDATA section, you use the following text:

```
A less than character &lt; and <b>bold text</b>.
```

Specifying HTML text

When you specify HTML text for a text control, the following rules apply:

- You cannot use a CDATA section directly in an inline `htmlText` property in an `<mx:Text>` tag. You must put the text in an `<mx:htmlText>` subtag, or in ActionScript code.
- Flex collapses consecutive white space characters, including return, space, and tab characters, in text that you specify in MXML property assignments or ActionScript outside of a CDATA section.
- If you specify the text in a CDATA section, you can use the text control's `condenseWhite` property to control whether Flex collapses white space. By default, the `condenseWhite` property is `false`, and Flex does not collapse white space.
- Use HTML <p> and <br> tags for breaks and paragraphs. In ActionScript CDATA sections You can also use \n escape characters.
- If your HTML text string is surrounded by single- or double-quotation marks because it is in an assignment statement (in other words, if it is not in an `<mx:htmlText>` tag), you must escape any uses of that quotation character in the string:
  - If you use double-quotation marks for the assignment delimiters, use " for the double-quotation mark (") character in your HTML. In ActionScript, you can also use the escape sequence \".

    NOTE: You do not need to escape double-quotation marks if you're loading text from an external file; it is only necessary if you're assigning a string of text in ActionScript.

  - If you use single-quotation marks for the assignment delimiters, use ' for the single-quotation mark character (') in your HTML. In ActionScript, you can also use the escape sequence \'.

- When you enter HTML-formatted text, you must include attributes of HTML tags in double- or single-quotation marks. Attribute values without quotation marks can produce unexpected results, such as improper rendering of text. You must follow the escaping rules for quotation marks within quotation marks, as described in "Escaping special characters in HTML text" on page 379.

The following example shows some simple HTML formatted text, using MXML and ActionScript to specify the text:

```
<?xml version="1.0"?>
<!-- textcontrols/HTMLFormattedText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500">
  <mx:Script><![CDATA[
     //The following is on one line.
     [Bindable]
     public var myHtmlText:String="This string contains <b>less than </b>,
&lt;, <b>greater than</b>, &gt;, <b>ampersand</b>, &amp;, and <b>double
quotation mark</b>, &quot;, characters.";
  ]]></mx:Script>

  <mx:Text id="htmltext2" width="450" htmlText="{myHtmlText}" />
  <mx:Text width="450">
     <mx:htmlText>
        <!-- The following is on one line. Line breaks would appear in the
output. -->
        <![CDATA[
           This string contains <b>less than</b>, &lt;, <b>greater than </
b>, &gt;, <b>ampersand</b>, &amp;, and <b>double quotation mark</b>,&quot;,
characters.
        ]]>
     </mx:htmlText>
  </mx:Text>
</mx:Application>
```

Each Text control displays the following text:

```
This string contains less than, <, greater than, >, ampersand, &, and double
quotation mark, " characters.
```

Escaping special characters in HTML text

The rules for escaping special characters in HTML text differ between CDATA sections and standard text.

In CDATA sections When you specify the `htmlText` string, the following rules apply:

- In ActionScript, but not in an `<mx:htmlText>` tag, you can use standard backslash escape sequences for special characters, such as \t for tab and \n for a newline character. You can also use the backslash character to escape many special characters, such as \' and \" for single- and double-quotation marks. You cannot use the combination \<, and a backslash before a return character has no effect on displayed text; it allows you to break the assignment statement across multiple text lines.
- In both ActionScript and the `<mx:htmlText>` tag, you can use HTML tags and numeric character entities; for example in place of \n, you can use a `<br>` tag.
- To include a left angle bracket (<), right angle bracket (>), or ampersand (&) character in displayed text, use the corresponding character entities: `&lt;`, `&gt;`, and `&amp;`, `respectively`. You can also use the `&quot;` and `&apos;` entities for single- and double-quotation marks. These are the only named character entities that Flash Player recognizes. Flash Player recognizes numeric entities, such as `&#165;` for the Yen mark (¥), however, it does not recognize the corresponding character entity, `&yen;`.

  The following code example uses the `htmlText` property to display formatted text:

```
<?xml version="1.0"?>
<!-- textcontrols/HTMLTags2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500">
  <mx:Text width="100%">
    <mx:htmlText><![CDATA[<p>This string contains a <b>less than</b>, &lt;.
       </p><p>This text is in a new paragraph.<br>This is a new line.</
p>]]>
    </mx:htmlText>
  </mx:Text>
</mx:Application>
```

This code displays the following text:

```
This string contains a less than, <.

This text is in a new paragraph.
This is a new line.
```

In standard text The following rules apply:

- You must use character entities, as described in "Using the htmlText property" on page 376, to use the left angle bracket (<), right angle bracket (>), or ampersand (&) character in HTML; for example, when you open a tag or start a character entity.

- You must use the `&` named entity combined with an HTML numeric character entity to display the less than character (use `&#060;`) and ampersand character (use `&#038;`). You can use the standard character entities, `>`, `&quot;`, and `&apos;`, for the greater than, double-quotation mark and single-quotation mark characters, respectively. For all other character entities, use numeric entity combinations, such as `&#165;`, for the Yen mark (¥).
- In ActionScript, but not in an `<mx:htmlText>` tag or inline `htmlText` property, you can use a backslash character to escape special characters, including the tab, newline, and quotation mark characters (but not the ampersand). In all cases, you can use (properly escaped) HTML tags and numeric character entities; for example in place of \n, you can use a `<br>` tag or `&#013;` entity.

Using tags in HTML text

When you use the `htmlText` property, you use a subset of HTML that is supported by the Flash Player. The Flash Player supports the following tags:

- Anchor tag (<a>)
- Bold tag (<b>)
- Break tag (<br>)
- Font tag (<font>)
- Image tag (<img>)
- Italic tag (<i>)
- List item tag (<li>)
- Paragraph tag (<p>)
- Text format tag (<textformat>)
- Underline tag (<u>)

Anchor tag (<a>)

The anchor `<a>` tag creates a hyperlink and supports the following attributes:

`href` Specifies the URL of the page to load in the browser. The URL can be absolute or relative to the location of the SWF file that is loading the page.

`target` Specifies the name of the target window to load the page into.

For example, the following HTML snippet creates the link “Go Home” to the Adobe Web site.

```
<a href='http://www.adobe.com' target='_blank'>Go Home</a>
```

You can also define `a:link`, `a:hover`, and `a:active` styles for anchor tags by using style sheets.

The `<a>` tag does *not* make the link text blue. You must apply formatting tags to change the text format. You can also define `a:link`, `a:hover`, and `a:active` styles for anchor tags by using style sheets.

The Label, Text, and TextArea controls can dispatch a `link` event when the user selects a hyperlink in the `htmlText` property. To generate the `link` event, prefix the hyperlink destination with `event:`, as the following example shows:

```
<?xml version="1.0"?>
<!-- textcontrols/LabelControlLinkEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    borderStyle="solid"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">

    <mx:Script>
        <![CDATA[
            import flash.events.TextEvent;

            public function linkHandler(event:TextEvent):void {
                myTA.text="link occured.";
                // Open the link in a new window.
                navigateToURL(new URLRequest(event.text), '_blank')
            }
        ]]>
    </mx:Script>

    <mx:Label selectable="true" link="linkHandler(event);">
        <mx:htmlText><![CDATA[<a href='event:http://www.adobe.com'>Adobe</
a>]]></mx:htmlText>
    </mx:Label>

    <mx:TextArea id="myTA"/>
</mx:Application>
```

The Label control must have the `selectable` property set to `true` to generate the `link` event.

When you use the `link` event, the event is generated and the text following `event:` in the hyperlink destination is included in the `text` property of the event object. However, the hyperlink is not automatically executed; you must execute the hyperlink from within your event handler. This allows you to modify the hyperlink, or even prohibit it from occurring, in your application.

Bold tag (‹b›)

The bold `<b>` tag renders text as bold. If you use embedded fonts, a boldface font must be available for the font or no text appears. If you use fonts that you expect to reside on the local system of your users, their system may approximate a boldface font if none exists, or it may substitute the normal font face instead of boldface. In either case, the text inside the bold tags will appear.

The following snippet applies boldface to the word *bold*:

```
This word is <b>bold</b>.
```

You cannot use the `</b>` end tag to override bold formatting that you set for all text in a control by using the `fontWeight` style.

Break tag (‹br›)

The break `<br>` tag creates a line break in the text. This tag has no effect in Label or TextInput controls.

The following snippet starts a new line after the word *line*:

```
The next sentence is on a new line.<br>Hello there.
```

Font tag (‹font›)

The `<font>` tag specifies the following font characteristics: color, face, and size.

The font tag supports the following attributes:

`color` Specifies the text color. You must use hexadecimal (`#FFFFFF`) color values. Other formats are not supported.

`face` Specifies the name of the font to use. You can also specify a list of comma-separated font names, in which case Flash Player chooses the first available font. If the specified font is not installed on the playback system, or isn't embedded in the SWF file, Flash Player chooses a substitute font. The following example shows how to set the font face.

`size` Specifies the size of the font in points. You can also use relative sizes (for example, +2 or -4).

The following example shows the use of the <font> tag:

```
<?xml version="1.0"?>
<!-- textcontrols/FontTag.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
backgroundGradientColors="[#FFFFFF, #FFFFFF]">
  <mx:TextArea height="100" width="250">
     <mx:htmlText>
        <![CDATA[
           You can vary the <font size='20'>font size</font>,<br><font
color="#0000FF">color</font>,<br><font face="CourierNew, Courier,
Typewriter">face</font>, or<br><font size="18" color="#FF00FF"face="Times,
Times New Roman, _serif">any combination of the three.</font>
        ]]>
     </mx:htmlText>
  </mx:TextArea>
</mx:Application>
```

This code results in the following output:

Image tag (<img>)

NOTE

The <img> tag is not fully supported in Flex 2, and might not work in some cases.

The image <img> tag lets you embed external JPEG, GIF, PNG, and SWF files inside text fields. Text automatically flows around images you embed in text fields. This tag is supported only in dynamic and input text fields that are multiline and wrap their text.

By default, Flash displays media embedded in a text field at full size. To specify dimensions for embedded media, use the <img> tag's height and width attributes.

In general, an image embedded in a text field appears on the line following the <img> tag. However, when the <img> tag is the first character in the text field, the image appears on the first line of the text field.

The <img> tag has one required attribute, src, which specifies the path to an image file. All other attributes are optional.

The `<img>` tag supports the following attributes:

`src` Specifies the URL to a GIF, JPEG, PNG, or SWF file. This attribute is required; all other attributes are optional. External files are not displayed until they have downloaded completely.

`align` Specifies the horizontal alignment of the embedded image within the text field. Valid values are `left` and `right`. The default value is `left`.

`height` Specifies the height of the image, in pixels.

`hspace` Specifies the amount of horizontal space that surrounds the image where no text appears. The default value is 8.

`id` Specifies the identifier for the imported image. This is useful if you want to control the embedded content with ActionScript.

`vspace` Specifies the amount of vertical space that surrounds the image where no text.

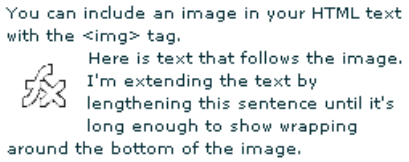
`width` Specifies the width of the image, in pixels.

appears. The default value is 8.

The following example shows the use of the `<img>` tag and how text can flow around the image:

```
<?xml version="1.0"?>
<!-- textcontrols/ImgTag.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
backgroundGradientColors="[#FFFFFF, #FFFFFF]" width="300" height="300">
  <mx:Text height="100%" width="100%">
     <mx:htmlText>
        <![CDATA[
          <p>You can include an image in your HTML text with the &lt;img&gt;
tag.</p><p><img src='../assets/bird.gif' width='30' height='30'
align='left' hspace='10' vspace='10'>Here is text that follows the image.
I'm extending the text by lengthening this sentence until it's long enough
to show wrapping around the bottom of the image.</p>
        ]]>
     </mx:htmlText>
  </mx:Text>
</mx:Application>
```

This code results in the following output:



Making hyperlinks out of embedded images

To make a hyperlink out of an embedded image, enclose the `<img>` tag in an `<a>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- textcontrols/ImgTagWithHyperlink.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500"
height="500" borderStyle="solid">
  <mx:TextArea width="100%" height="100%">
     <mx:htmlText>
        <![CDATA[
           <a href='http://www.adobe.com'><img src='../assets/bird.gif'/></
a>Click the image to go to the Adobe home page.
        ]]>
     </mx:htmlText>
  </mx:TextArea>
</mx:Application>
```

When the user moves the mouse pointer over an image that is enclosed by `<a>` tags, the mouse pointer changes to a hand icon, as with standard hyperlinks. Interactivity, such as mouse clicks and key presses, do not register in SWF files that are enclosed by `<a>` tags.

Italic tag (<i>)

The italic `<i>` tag displays the tagged text in italic font. If you're using embedded fonts, an italic font must be available or no text appears. If you use fonts that you expect to reside on the local system of your users, their system may approximate an italic font if none exists, or it may substitute the normal font face instead of italic. In either case, the text inside the italic tags appears.

The following snippet applies italic font to the word *italic*:

```
The next word is in <i>italic</i>.
```

You cannot use the `</i>` end tag to override italic formatting that you set for all text in a control by using the `fontStyle` style.

List item tag (<li>)

The list item <li> tag ensures that the text that it encloses starts on a new line with a bullet in front of it. You cannot use it for any other type of HTML list item. The ending tag ensures a line break (but generates a single line break). Unlike in HTML, you do not surround <li> tags in <ul> tags. For example, the following Flex code generates a bulleted list with two items:

```
<?xml version="1.0"?>
<!-- textcontrols/BulletedListExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500">
  <mx:Text width="100%">
     <mx:htmlText >
        <![CDATA[
           <p>This is a bulleted list:<li>First Item</li><li>Second Item</
li></p>
        ]]>
     </mx:htmlText>
  </mx:Text>
</mx:Application>
```

NOTE

In Flex 2, the <li> tag does not work properly with Label controls. With TextInput controls, it must be put before the first character in the text.

Paragraph tag ( )

The paragraph tag creates a new paragraph. The opening tag does *not* force a line break, but the closing tag does. Unlike in HTML, the tag does not force a double space between paragraphs; the spacing is the same as that generated by the <br> tag.

The tag supports the following attribute:

`align` Specifies alignment of text in the paragraph; valid values are `left`, `right`, and `center`.

The following snippet generates two centered paragraphs:

```
<p align="center">This is a first centered paragraph</p>
<p align="center">This is a second centered paragraph</p>
```

Text format tag (<textformat>)

The text format <textformat> tag lets you use a subset of paragraph formatting properties of the TextFormat class in HTML text fields, including line leading, indentation, margins, and tab stops. You can combine text format tags with the built-in HTML tags. The text format tag supports the following attributes:

- `blockindent` Specifies the indentation, in points, from the left margin to the text in the <textformat> tag body.

- `indent` Specifies the indentation, in points, from the left margin or the block indent, if any, to the first character in the `<textformat>` tag body.
- `leading` Specifies the amount of leading (vertical space) between lines.
- `leftmargin` Specifies the left margin of the paragraph, in points.
- `rightmargin` Specifies the right margin of the paragraph, in points.
- `tabstops` Specifies custom tab stops as an array of nonnegative integers.

Underline tag (<u>)

The underline `<u>` tag underlines the tagged text.

The following snippet underlines the word *underlined*:

```
The next word is <u>underlined</u>.
```

You cannot use the `</u>` end tag to override underlining that you set for all text in a control by using the `textDecoration` style.

Selecting and modifying text

You can select and modify text in TextArea, TextInput and RichTextEditor controls, as described in the following sections. (To change a Label or Text control's text, assign a new value to the control's `text` or `HTMLtext` property. For more information on the `HTMLText` property, see "Using the htmlText property" on page 376.

Selecting text

The Flex editable controls provide properties and methods to select text regions and get selections, as the following sections describe. You can modify the contents of the selection as described in "Modifying text" on page 388.

Creating a selection

The TextInput and TextArea controls, including the RichTextEditor control's TextArea subcontrol, provide the following text selection properties and method:

- `setSelection()` method selects a range of text. You specify the zero-based indexes of the start character and the position immediately *after* the last character in the text.
- `selectionBeginIndex` and `selectionEndIndex` set or return the zero-based location in the text of the start and position immediately *after* the end of a selection.

To select the first 10 characters of the myTextArea TextArea control, for example, use the following method:

```
myTextArea.setSelection(0, 10);
```

To change the last character of this selection to be the twenty-fifth character in the TextArea control, use the following statement:

```
myTextArea.endIndex=25;
```

To select text in a RichTextEditor control, use the control's TextArea subcontrol, which you access by using the `textArea` id. To select the first 10 characters in the myRTE RichTextEditor control, for example, use the following code:

```
myRTE.textArea.setSelection(0, 10);
```

Getting a selection

You get a text control's selection by getting a TextRange object with the selected text. You can then use the TextRange object to modify the selected text, as described in "Modifying text" on page 388. The technique you use to get the selection depends on the control type, as the following sections describe.

To get the selection in a TextArea or TextInput control

Use the TextRange class constructor to get a TextRange object with the currently selected text in a TextArea or TextInput control. For example, to get the current selection of the myTextArea control, use the following line:

```
var mySelectedTextRange:TextRange = new TextRange(myTextArea, true);
```

The second parameter, `true`, tells the constructor to return a TextRange object with the selected text.

To get the selection in a RichTextEditor control

Use the `selection` read-only property of the RichTextEditor to get a TextRange object with the currently selected text in its TextArea subcontrol. You can use the TextRange object to modify the selected text, as described in "Modifying text". For example, to get the current selection of the MyRTE RichTextEditor control, us the following line:

```
public var mySelectedTextRange:TextRange = myRTE.selection;
```

Modifying text

You use the TextRange class to modify the text in a TextArea, TextInput, or RichTextEditor control. This class lets you affect the following text characteristics:

- `text` or `htmltext` property contents

- text color, decoration (underlining), and alignment
- font family, size, style (italics), and weight (bold)
- URL of an HTML <a> link.

Getting a TextRange object

To get a TextRange object you use the following techniques:

- Get a TextRange object that contains the current text selection, as described in "Getting a selection" on page 388.
- Create a TextRange object that contains a specific range of text.

To create a TextRange object with a specific range of text, use a TextRange constructor with the following format:

```
new TextRange(control, modifiesSelection, beginIndex, endIndex)
```

Specify the control that contains the text, whether the TextRange object corresponds to a selection (that is, represents and modifies selected text), and the zero-based indexes in the text of the first and last character of the range. As a general rule, do not use the TextRange constructor to set a selection; use the `setSelection()` method, as described in "Creating a selection" on page 387. For this reason, the second parameter should always be `false` when you specify the begin and end indexes.

To get a TextRange object with the fifth through twenty-fifth characters of a TextArea control named myTextArea, for example, use the following line:

```
var myTARange:TextRange = new TextRange(myTextArea, false, 4, 25);
```

Changing text

After you get a TextRange object, use its properties to modify the text in the range. The changes you make to the TextRange appear in the text control.

You can get or set the text in a TextRange object as HTML text or as a plain text, independent of any property that you might have used to initially set the text. If you created a TextArea control, for example, and set its `text` property, you can use the TextRange `htmlText` property to get and change the text. The following example shows this usage, and shows using the TextRange class to access a range of text and change its properties. It also shows using String properties and methods to get text indexes.

```
<?xml version="1.0"?>
<!-- textcontrols/TextRangeExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
     height="500">
  <mx:Script><![CDATA[
     import mx.controls.textClasses.TextRange

     public function alterText():void {
        // Create a TextRange object starting with "the" and ending at the
        // first period. Replace it with new formatted HTML text.
        var tr1:TextRange = new TextRange(ta1, false,
ta1.text.indexOf("the", 0), ta1.text.indexOf(".", 0));
        tr1.htmlText="<i>italic HTML text</i>"

        // Create a TextRange object with the remaining text.
        // Select the text and change its formatting.
       var tr2:TextRange = new TextRange(ta1, true, ta1.text.indexOf("It",
0), ta1.text.length-1);
        tr2.color=0xFF00FF;
        tr2.fontSize=18;
        tr2.fontStyle = "italic"; // any other value turns italic off
        tr2.fontWeight = "bold"; // any other value turns bold off
        ta1.setSelection(0, 0);
     }
  ]]></mx:Script>

  <mx:TextArea id="ta1" fontSize="12" fontWeight="bold" width="100%"
height="100">
     <mx:text>
        This is a test of the emergency broadcast system. It is only a test.
     </mx:text>
  </mx:TextArea>
  <mx:Button label="Alter Text" click="alterText();"/>
</mx:Application>
```

Example: changing selected text in a RichTextEditor control

The following example shows how you can use the `selectedText` property of the RichTextEditor control to get a TextRange when a user selects some text, and use TextRange properties to get and change the characteristics of the selected text. To use the example, select a range of text with your mouse. When you release the mouse button, the string "This is replacement text. ", formatted in fuchsia Courier 20-point font replaces the selection and the text area reports on the original and replacement text.

```
<?xml version="1.0"?>
<!-- textcontrols/TextRangeSelectedText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="500">
  <mx:Script><![CDATA[
     import mx.controls.textClasses.TextRange;

     //The following text must be on a single line.
     [Bindable]
     public var htmlData:String="<textformat leading='2'><p
align='center'><b><font size='20'>HTML Formatted Text</font></b></p></
textformat><br><textformat leading='2'><p align='left'><font face='_sans'
size='12' color='#000000'>This paragraph contains <b>bold</b>, <i>italic</
i>, <u>underlined</u>, and <b><i><u>bold italic underlined </u></i></
b>text. </font></p></textformat><br><p><u><font face='arial' size='14'
color='#ff0000'>This a red underlined 14-point arial font with no alignment
set.</font></u></p><p align='right'><font face='verdana' size='12'
color='#006666'><b>This a teal bold 12-pt. Verdana font with alignment set
to right.</b></font></p>";

     public function changeSelectionText():void {
        //Get a TextRange with the selected text and find its length.
        var sel:TextRange = rte1.selection;
        var selLength:int = sel.endIndex - sel.beginIndex;
        //Do the following only if the user made a selection.
        if (selLength) {
           //Display the selection size and font color, size, and family.
           t1.text="Number of characters selected: " + String(selLength);
           t1.text+="\n\nOriginal Font Family: " + sel.fontFamily;
           t1.text+="\nOriginal Font Size: " + sel.fontSize;
           t1.text+="\nOriginal Font Color: " + sel.color;
           //Change font color, size, and family and replace selected text.
           sel.text="This is replacement text. "
           sel.color="fuchsia";
           sel.fontSize=20;
           sel.fontFamily="courier"
           //Show the new font color, size, and family.
           t1.text+="\n\nNew text length: " + String(sel.endIndex -
```

```
sel.beginIndex);
            t1.text+="\nNew Font Family: " + sel.fontFamily;
            t1.text+="\nNew Font Size: " + sel.fontSize;
            t1.text+="\nNew Font Color: " + sel.color;
        }
    }
  ]]></mx:Script>

  <!-- The text area. When you release the mouse after selecting text,
     it calls the func1 function. -->
  <mx:RichTextEditor id="rte1" htmlText="{htmlData}" width="100%"
height="100%" mouseUp="changeSelectionText()"/>
  <mx:TextArea editable="false" id="t1" fontSize="12" fontWeight="bold"
width="300" height="180"/>

</mx:Application>
```

Label control

The Label control is a noneditable single-line text label. It has the following characteristics:

- The user cannot change the text, but the application can modify it.
- You can specify text formatting by using styles or HTML text.
- You can control the alignment and sizing.
- The control is transparent and does not have a `backgroundColor` property, so the background of the component's container shows through.
- The control has no borders, so the label appears as text written directly on its background.
- The control cannot take the focus.

For complete reference information, see Label in *Adobe Flex 2 Language Reference*.

To create a multiline, noneditable text field, use a Text control. For more information, see "Text control" on page 395. To create user-editable text fields, use TextInput or TextArea controls. For more information, see "TextInput control" on page 393 and "TextArea control" on page 397.

The following image shows a Label control:

For the code used to create this sample, see "Creating a Label control".

Creating a Label control

You define a Label control in MXML by using the `<mx:Label>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/LabelControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="150"
height="80" borderStyle="solid" backgroundGradientColors="[#FFFFFF,
#FFFFFF]">
  <mx:Label text="Label1"/>
</mx:Application>
```

You use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information on using these properties, see "Using the text property" on page 371 and "Using the htmlText property" on page 376.

Sizing a Label control

The Label control has the following default sizing properties:

Property	Default value
Default size	Width and height large enough for the text
Minimum size	0
Maximum size	10000 by 10000 pixels

If you do not specify a width, the Label control automatically resizes when you change the value of the `text` or `htmlText` property.

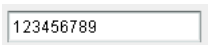
If you explicitly size a Label control so that it is not large enough to accommodate its text, the text is truncated and terminated by an ellipses (...). The full text displays as a tooltip when you move the mouse over the Label control. If you also set a tooltip by using the `tooltip` property, the tooltip is displayed rather than the text.

TextInput control

The TextInput control is a single-line text field that is optionally editable. The TextInput control supports the HTML rendering capabilities of Adobe Flash Player.

For complete reference information, see TextInput in *Adobe Flex 2 Language Reference*.

The following image shows a TextInput control:



To create a multiline, editable text field, use a TextArea control. For more information, see "TextArea control" on page 397. To create noneditable text fields, use Label and Text controls. For more information, see "Label control" on page 392 and "Text control" on page 395.

The TextInput control does not include a label, but you can add one using a Label control or by nesting the TextInput control in a FormItem container in a Form layout container, as shown in the example in "About text controls" on page 369. TextInput controls dispatch `change`, `textInput`, and `enter` events.

If you disable a TextInput control, it displays its contents in a different color, represented by the `disabledColor` style. You can set a TextInput control's `editable` property to `false` to prevent editing of the text. You can set a TextInput control's `displayAsPassword` property to conceal the input text by displaying characters as asterisks.

Creating a TextInput control

You define a TextInput control in MXML using the `<mx:TextInput>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/TextInputControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:TextInput id="text1" width="100"/>
</mx:Application>
```

Just as you can for the Label control, you use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see "Using the text property" on page 371 and "Using the htmlText property" on page 376.

Sizing a TextInput control

The TextInput control has the following default sizing properties:

Property	Default value
Default size	The size of the text with a default minimum size of 22 pixels high and 160 pixels wide

Property	Default value
Minimum size	0
Maximum size	10000 by 10000 pixels

If you do not specify a width, the TextInput control automatically resizes when you change the value of the `text` or `htmlText` property. It does not resize in response to typed user input.

Binding to a TextInput control

In some cases, you might want to bind a variable to the text property of a TextInput control so that the control represents a variable value, as the following example shows:

```
<?xml version="1.0"?>
<!-- textcontrols/BoundTextInputControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     [Bindable]
     public var myProp:String="This is the initial myPropString.";
  ]]></mx:Script>
  <mx:TextInput text="{myProp}"/>
</mx:Application>
```

In this example, the TextInput control displays the value of the myProp variable. Remember that you must use the `[Bindable]` metadata tag if the variable changes value and the control must track the changed values; also, the compiler generates warnings if you do not use this metadata tag.

Text control

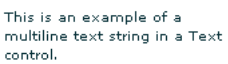
The Text control displays multiline, noneditable text. The control has the following characteristics:

- The user cannot change the text, but the application can modify it.
- The control does not support scroll bars. If the text exceeds the control size users can use keys to scroll the text
- The control is transparent so that the background of the component's container shows through
- The control has no borders, so the label appears as text written directly on its background.
- The control supports HTML text and a variety of text and font styles.
- The text always word-wraps at the control boundaries, and is always aligned to the top of the control.

For complete reference information, see Text in *Adobe Flex 2 Language Reference*.

To create a single-line, noneditable text field, use the Label control. For more information, see "Label control" on page 392. To create user-editable text fields, use the TextInput or TextArea controls. For more information, see "TextInput control" on page 393 and "TextArea control" on page 397.

The following image shows an example of a Text control with a with of 175 pixels:



Creating a Text control

You define a Text control in MXML using the `<mx:Text>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/TextControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Text width="175" text="This is an example of a multiline text string
in a Text control."/>
</mx:Application>
```

You use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see "Using the text property" on page 371 and "Using the htmlText property" on page 376.

This control does not support a `backgroundColor` property; its background is always the background of the control's container.

Sizing a Text control

The Text control has the following default sizing properties:

Property	Default value
Default size	Flex sizes the control to fit the text, with the control width is long enough to fit the longest line of text, and the height tall enough to fit the number of lines. If you do not specify a pixel width, the height is determined by the number of explicit line breaks in the text string. If the text length changes, the control resizes to fit the new text.
Minimum size	0
Maximum size	10000 by 10000 pixels

Flex sizes the Text control as follows:

- If you specify a pixel value for both the `height` and `width` properties, any text that exceeds the size of the control is clipped at the border.
- If you specify an explicit pixel width, but no height, Flex wraps the text to fit the width and calculates the height to fit the required number of lines.
- If you specify a percentage-based width and no height, Flex does *not* wrap the text, and the height equals the number of lines as determined by the number of Return characters.
- If you specify only a height and no width, the height value does not affect the width calculation, and Flex sizes the control to fit the width of the maximum line.

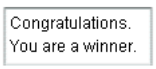
As a general rule, if you have long text, you should specify a pixel-based `width` property. If the text might change and you want to ensure that the Text control always takes up the same space in your application, set explicit `height` and `width` properties that fit the largest expected text.

TextArea control

The TextArea control is a multiline, editable text field with a border and optional scroll bars. The TextArea control supports the HTML and rich text rendering capabilities of Flash Player. The TextArea control dispatches `change` and `textInput` events.

For complete reference information, see TextArea in *Adobe Flex 2 Language Reference*.

The following image shows a TextArea control:



To create a single-line, editable text field, use the TextInput control. For more information, see "TextInput control" on page 393. To create noneditable text fields, use the Label and Text controls. For more information, see "Label control" on page 392 and "Text control" on page 395.

If you disable a TextArea control, it displays its contents in a different color, represented by the `disabledColor` style. You can set a TextArea control's `editable` property to `false` to prevent editing of the text. You can set a TextArea control's `displayAsPassword` property to conceal input text by displaying characters as asterisks.

Creating a TextArea control

You define a TextArea control in MXML using the `<mx:TextArea>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/TextAreaControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:TextArea id="textConfirm" width="300" height="100" text="Please enter
your thoughts here."/>
</mx:Application>
```

Just as you can for the Text control, you use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see "Using the text property" on page 371 and "Using the htmlText property" on page 376.

Sizing the TextArea control

The TextArea control has the following default sizing properties:

Property	Default value
Default size	160 pixels for the `width` property 44 pixels for the `height` property
Minimum size	0
Maximum size	10000 by 10000 pixels

The TextArea control does not resize to fit the text that it contains. If the new text exceeds the capacity of the TextArea control and the `horizontalScrollPolicy` is `true` (the default value), the control adds a scrollbar.

RichTextEditor control

The RichTextEditor control lets users enter, edit, and format text. Users apply text formatting and URL links using subcontrols that are located at the bottom of the RichTextEditor control.

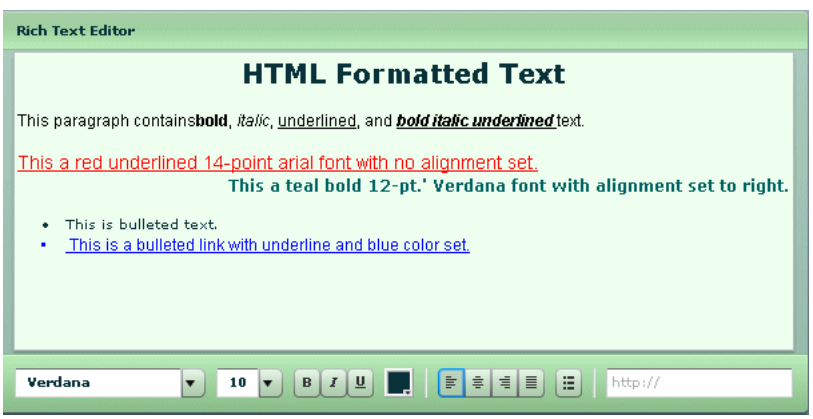
For complete reference information, see RichTextEditor in *Adobe Flex 2 Language Reference*.

About the RichTextEditor control

The RichTextEditor control consists a Panel control with two direct children:

- A TextArea control in which users can enter text
- A tool bar container with format controls that let a user specify the text characteristics. Users can use the tool bar subcontrols to apply the following text characteristics:
- Font family
- Font size
- Any combination of bold, italic and underline font styles
- Text color
- Text alignment: left, center, right, or justified
- Bullets
- URL links

The following image shows a RichTextEditor control with some formatted text:



For the source for this example, see "Creating a RichTextEditor control" on page 400.

You use the RichTextEditor interactively as follows:

- Text that you type is formatted as specified by the control settings.
- To apply new formatting to existing text, select the text and set the controls to the required format.
- To create a link, select a range of text, enter the link target in the text box on the right, and press Enter. You can only specify the URL; the link always opens in a _blank target. Also, creating the link does not change the appearance of the link text; you must separately apply any color and underlining.

- You can cut, copy, and paste rich text within and between Flash HTML text fields, including the RichTextEditor control's TextArea subcontrol, using the normal keyboard commands. You can copy and paste plain text between the TextArea and any other text application, such as your browser or a text editor.

Creating a RichTextEditor control

You define a RichTextEditor control in MXML using the `<mx:RichTextEditor>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<!-- textcontrols/RichTextEditorControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:RichTextEditor id="myRTE" text="Congratulations, winner!" />
</mx:Application>
```

You can use the `text` property to specify an unformatted text string, or the `htmlText` property to specify an HTML-formatted string. For more information on using these properties, see "Using the text property" on page 371, and "Using the htmlText property" on page 376. For information on selecting, replacing, and formatting text that is in the control, see "Selecting and modifying text" on page 387.

The following example shows the code used to create the image in "About the RichTextEditor control" on page 399:

```
<?xml version="1.0"?>
<!-- textcontrols/RichTextEditorControlWithFormattedText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="700"
height="400">
  <!-- The HTML text string used to populate the RichTextEditor control's
     TextArea subcontrol. The text is on a single line. -->
  <mx:Script><![CDATA[
     [Bindable]
     public var htmlData:String="<textformat leading='2'><p
align='center'><b><font size='20'>HTML Formatted Text</font></b></p></
textformat><br><textformat leading='2'><p align='left'><font face='_sans'
size='12' color='#000000'>This paragraph contains<b>bold</b>, <i>italic</
i>, <u>underlined</u>, and <b><i><u>bold italic underlined </u></i></
b>text.</font></p></textformat><br><p><u><font face='arial' size='14'
color='#ff0000'>This a red underlined 14-point arial font with no alignment
set.</font></u></p><p align='right'><font face='verdana' size='12'
color='#006666'><b>This a teal bold 12-pt.' Verdana font with alignment set
to right.</b></font></p><br><li>This is bulleted text.</li><li><font
face='arial' size='12' color='#0000ff'><u> <a href='http://
www.adobe.com'>This is a bulleted link with underline and blue color set.</
a></u></font></li>";
  ]]></mx:Script>

  <!-- The RichTextEditor control. To reference a subcontrol prefix its ID
with the RichTextEditor control ID. -->
  <mx:RichTextEditor id="rte1"
     backgroundColor="#ccffcc"
     width="605"
     headerColors="[#88bb88, #bbeebb]"
     footerColors="[#bbeebb, #88bb88]"
     title="Rich Text Editor"
     htmlText="{htmlData}"
     initialize="rte1.textArea.setStyle('backgroundColor', '0xeeffee')"
  />
</mx:Application>
```

Sizing the RichTextEditor control

The RichTextEditor control has the following default sizing properties:

Property	Default value
Default size	325 pixels wide by 300 pixels high
Minimum size	220 pixels wide by 200 pixels high
Maximum size	10000 by 10000 pixels

The control does not resize in response to the size of the text in the TextArea control. If the text exceeds the viewable space, by default, the TextArea control adds scroll bars. If you specify a value for either the `height` or `width` property but not both, the control uses the default value for the property that you do not set.

If you set a `width` value that results in a width less than 605 pixels wide, The RichTextEditor control stacks the subcontrols in rows.

Programming RichTextEditor subcontrols

Your application can control the settings of any of the RichTextEditor subcontrols, such as the TextArea, the ColorPicker, or any of the ComboBox or Button controls that control text formatting. To refer to a RichTextEditor subcontrol, prefix the requested control's ID, as listed in the RichTextEditor entry in *Adobe Flex 2 Language Reference*, with the RichTextEditor control ID. For example, to refer to the ColorPicker control in a RichTextEditor control that has the ID rte1, use rte1.colorPicker.

Inheritable styles that you apply directly to a RichTextEditor control affect the underlying Panel control and the subcontrols. Properties that you apply directly to a RichTextEditor control affect the underlying Panel control only.

Setting RichTextEditor subcontrol properties and styles

The following simple code example shows how you can set and change the properties and styles of the RichTextEditor control and its subcontrols. This example uses styles that the RichTextEditor control inherits from the Panel class to set the colors of the Panel control header and the tool bar container, and sets the TextArea control's background color in the RichTextEditor control's creationComplete event member. When users click the buttons, their click event listeners change the TextArea control's background color and the selected color of the ColorPicker control.

```
<?xml version="1.0"?>
<!-- textcontrols/RTESubcontrol.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
     height="420">
  <!-- The RichTextEditor control. To set the a subcontrol's style or
property, fully qualify the control ID. The footerColors style sets the
ControlBar colors. -->
  <mx:RichTextEditor id="rte1"
     backgroundColor="#ccffcc"
     headerColors="[#88bb88, #bbeebb]"
     footerColors="[#bbeebb, #88bb88]"
     title="Rich Text Editor"

creationComplete="rte1.textArea.setStyle('backgroundColor','0xeeffee')"
     text="Simple sample text"
  />

  <!-- Button to set a white TextArea background. -->
  <mx:Button
     label="Change appearance"
     click="rte1.textArea.setStyle('backgroundColor',
'0xffffff');rte1.colorPicker.selectedIndex=27;"
  />

  <!-- Button to reset the display to its original appearance. -->
  <mx:Button
     label="Reset Appearance"
     click="rte1.textArea.setStyle('backgroundColor',
'0xeeffee');rte1.colorPicker.selectedIndex=0;"
  />
</mx:Application>
```

Removing and adding RichTextEditor subcontrols.

You can remove any of the standard RichTextEditor subcontrols, such as the alignment buttons. You can also add your own subcontrols, such as a button that pops up a find-and-replace dialog box.

To remove an existing subcontrol

1. Create a function that calls the `removeChildAt` method of the editor's tool bar Container subcontrol, specifying the control to remove.
2. Call the method in the RichTextEditor control's `initialize` event listener.

The following example removes the alignment buttons from a RichTextEditor control, and shows the default appearance of a second RichTextEditor control:

```
<?xml version="1.0"?>
<!-- textcontrols/RTERemoveAlignButtons.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     public function removeAlignButtons():void {
        rt1.toolbar.removeChild(rt1.alignButtons);
     }
  ]]></mx:Script>

  <mx:RichTextEditor id="rt1" title="RichTextEditor With No Align Buttons"
creationComplete="removeAlignButtons()"/>

  <mx:RichTextEditor id="rt2" title="Default RichTextEditor"/>

</mx:Application>
```

To add a new subcontrol:

1. Create an ActionScript function that defines the subcontrol. Also create any necessary methods to support the control's function.
2. Call the method in the RichTextEditor control's `initialize` event listener, as in the following tag:

   ```
   <mx:RichTextEditor id="rt" initialize="addMyControl()"
   ```

The following example adds a find-and-replace dialog box to a RichTextEditor control. It consists of two files: the application, and a custom TitleWindow control that defines the find-and-replace dialog (which also performs the find-and-replace operation on the text). The application includes a function that adds a button to pop up the TitleWindow, as follows:

```
<?xml version="1.0"?>
<!-- textcontrols/CustomRTE.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
     width="600" height="100%">
  <mx:Script>
     <![CDATA[
        import mx.controls.*;
        import mx.containers.*;
        import flash.events.*;
        import mx.managers.PopUpManager;
        import mx.core.IFlexDisplayObject;

        // The variable for the pop-up dialog box.
        public var w:IFlexDisplayObject;

        // Add the Find/Replace button to the Rich Text Editor control's
        // tool bar container.
        public function addFindReplaceButton():void {
           var but:Button = new Button();
           but.label = "Find/Replace";
           but.addEventListener("click",findReplaceDialog);
           rt.toolbar.addChild(but);
        }

        // The event listener for the Find/Replace button's click event
        // creates a pop-up with a MyTitleWindow custom control.
        public function findReplaceDialog(event:Event):void {
           var w:MyTitleWindow = MyTitleWindow(PopUpManager.createPopUp
            (this, MyTitleWindow, true));
           w.height=200;
           w.width=340;
           // Pass the a reference to the textArea subcontrol
           // so that the custom control can replace the text.
           w.RTETextArea = rt.textArea;
           PopUpManager.centerPopUp(w);
        }
     ]]>
  </mx:Script>

<mx:RichTextEditor id="rt" width="95%" title="RichTextEditor"
     text="This is a short text."
     initialize="addFindReplaceButton()"/>

</mx:Application>
```

The following MyTitleWindow.mxml file defines the custom myTitleWindow control that contains the find-and-replace interface and logic:

```
<?xml version="1.0"?>
<!-- A TitleWindow that displays the X close button. Clicking the close
button only generates a CloseEvent event, so it must handle the event to
close the control. -->
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml" title="Find/
Replace" showCloseButton="true" close="closeDialog();">
<mx:Script>
<![CDATA[
import mx.controls.TextArea;
import mx.managers.PopUpManager;

// Reference to the RichTextArea textArea subcontrol.
// It is set by the application findReplaceDialog method
// and used in the replaceAndClose method, below.
public var RTETextArea:TextArea;

// The event handler for the Replace button's click event.
// Replace the text in the RichTextEditor TextArea and
// close the dialog box.
public function replaceAndClose():void{
RTETextArea.text = RTETextArea.text.replace(ti1.text, ti2.text);
PopUpManager.removePopUp(this);
}

// The event handler for the TitleWindow close button.
public function closeDialog():void {
PopUpManager.removePopUp(this);
}

]]>
</mx:Script>

<!-- The TitleWindow subcontrols: the find and replace inputs,
their labels, and a button to initiate the operation. -->
<mx:Label text="Find what:"/>
<mx:TextInput id="ti1"/>
<mx:Label text="Replace with:"/>
<mx:TextInput id="ti2"/>
<mx:Button label="Replace" click="replaceAndClose();"/>

</mx:TitleWindow>
```

CHAPTER 11

Using Menu-Based Controls

11

Several Adobe Flex framework controls create or interact with menus. This topic describes these controls and discusses how to use them.

Contents

About menu-based controls

Flex framework includes three controls that present hierarchical data in a cascading menu format. All menu controls can have icons and labels for each menu item, and dispatch events of the mx.events.MenuEvent class in response to user actions. You can use the following menu-based controls:

Menu control A visual menu that can have cascading submenus. You typically display a menu control in response to some user action, such as clicking a button. You cannot define a Menu control by using an MXML tag; you must define it, display it, and hide it using ActionScript.

MenuBar control A horizontal bar of menu items. Each item in the menu bar can have a submenu that displays when you click the MenuBar item. The MenuBar control is effectively a static (non-pop-up) menu that displays the top level of the menu as the bar items.

PopUpMenuButton control A subclass of the PopUpButton control that displays a Menu control when you click the secondary button. The primary button label changes when you select an item from the pop-up menu.

Defining menu structure and data

All menu-based controls use data providers with the following characteristics to specify the structure and contents of the menus:

- The data providers are often hierarchical, but you can also have a single-level menu.
- Individual menu items include fields that determine the menu item appearance and behavior. Menu-based controls support fields that define the label text, an icon, the menu item type, and item status. For information on meaningful fields, see "Specifying and using menu entry information" on page 409.

About menu data providers

The `dataProvider` property of a menu-based control specifies an object that defines the structure and contents of the menu. If a menu's contents are dynamic, you change the menu by modifying its data provider.

Menu-based controls typically get their data from hierarchical data providers, such as nested arrays of objects or XML. If the menu represents dynamically changing data, you use an object that implements the ICollectionView interface, such as ArrayCollection or XMLListCollection.

Menu-based controls use a data descriptor to parse and manipulate the data provider contents. By default, menu-based controls use a DefaultDataDescriptor class descriptor, but you can create your own class and specify it in the Menu control's `dataDescriptor` property.

The DefaultDataDescriptor class supports the following types of data:

XML A string that contains valid XML text, or any of the following objects containing valid E4X format XML data: an `<mx:XML>` or `<mx:XMLList>` compile-time tag, or an XML or XMLList object.

Other objects An array of items, or an object that contains an array of items, where a node's children are contained in an item named `children`. You can also use the `<mx:Model>` compile-time tag to create nested objects that support data binding, but you must follow the structure defined in "Using the <mx:Model> tag with Tree and menu-based controls" on page 200.

Collections An object that implements the ICollectionView interface (such as the ArrayCollection or XMLListCollection classes) and whose data source conforms to the structure specified in either of the previous bullets. The DefaultDataDescriptor class includes code to handle collections efficiently. Always use a collection as the data provider if the data in the menu changes dynamically; otherwise, the Menu displays obsolete data.

For more information on hierarchical objects and data descriptors, including a detailed description of the formats supported by the DefaultDataDescriptor, see "Data descriptors and hierarchical data provider structure" on page 197.

As with all data-driven controls, if the data provider contents can change dynamically, and you want the Menu to update with the changes, ensure that the data source is a collection, such as an ArrayCollection or XMLListCollection object. To modify the menu, change the underlying collection, and the menu will update its appearance accordingly.

Node (menu item) tags in the XML data can have any name. Many examples in this topic use tags such as `<node>` for all menu items, or `<menuItem>` for top-level items and `<subMenuItem>` for submenu items, but it might be more realistic to use tag names that identify the data, such as `<person>`, `<address>`, and so on. The menu-handling code reads through the XML and builds the display hierarchy based on the nested relationship of the nodes. For more information, see "Specifying and using menu entry information".

Most menus have multiple items at the top level, not a single root item. XML objects, such as the XML object created by the `<mx:XML>` tag, must have a single root node. To display a menu that uses a data provider that has a root that you do not want to display, set the Menu, PopUpMenuButton, or MenuBar `showRoot` property to `false`.

Specifying and using menu entry information

Information in menu-based control data providers determines how each menu entry appears and is used. To access or change the menu contents, you modify the contents of the data provider.

The menu-based classes use the methods of the IMenuDataDescriptor class to access and manipulate information in the data provider that defines the menu behavior and contents. Flex provides a DefaultDataDescriptor class that implements these interface. The menu-based controls use the DefaultDataDescriptor class if you do not set the `dataDescriptor` property. This section describes the menu information you can provide, and the data provider fields and values you use when using DefaultDataDescriptor class.

Menu entry types

Each data provider entry can specify an item type and type-specific information about the menu item. Menu-based classes support the following item types (`type` field values):

normal (the default) Selecting an item with the `normal` type triggers a `change` event, or, if the item has children, opens a submenu.

check Selecting an item with the `check` type toggles the menu item's `toggled` property between `true` and `false` values. When the menu item is in the `true` state, it displays a check mark in the menu next to the item's label.

radio Items with the `radio` type operate in groups, much like RadioButton controls; you can select only one radio menu item in each group at a time. The example in this section defines three submenu items as radio buttons within the group "one".

When a radio button is selected, the radio item's `toggled` property is set to `true`, and the `toggled` properties of all other radio items in the group are set to `false`. The Menu control displays a solid circle next to the radio button that is currently selected. The `selection` property of the radio group is set to the label of the selected menu item.

separator Items with the `separator` type provide a simple horizontal line that divides the items in a menu into different visual groups.

Menu attributes

Menu items can specify several attributes that determine how the item is displayed and behaves. The following table lists the attributes you can specify, their data types, their purposes, and how the data provider must represent them if the menu uses the DefaultDataDescriptor class to parse the data provider:

Attribute	Type	Description
`enabled`	Boolean	Specifies whether the user can select the menu item (`true`), or not (`false`). If not specified, Flex treats the item as if the value were `true`. If you use the default data descriptor, data providers must use an `enabled` XML attribute or object field to specify this characteristic.
`groupName`	String	(Required, and meaningful, for `radio` type only) The identifier that associates radio button items in a radio group. If you use the default data descriptor, data providers must use a `groupName` XML attribute or object field to specify this characteristic.
`icon`	Class	Specifies the class identifier of an image asset. This item is not used for the `check`, `radio`, or `separator` types. You can use the `checkIcon` and `radioIcon` styles to specify the icons used for radio and check box items that are selected. The menu's `iconField` or `iconFunction` property determines the name of the field in the data that specifies the icon, or a function for determining the icons.

Attribute	Type	Description
`label`	String	Specifies the text that appears in the control. This item is used for all menu item types except `separator`. The menu's `labelField` or `labelFunction` property determines the name of the field in the data that specifies the label, or a function for determining the labels. (If the data provider is in E4X XML format, you must specify one of these properties to display a label.) If the data provider is an array of strings, Flex uses the string value as the label.
`toggled`	Boolean	Specifies whether a `check` or `radio` item is selected. If not specified, Flex treats the item as if the value were `false` and the item is not selected. If you use the default data descriptor, data providers must use a `toggled` XML attribute or object field to specify this characteristic.
`type`	String	Specifies the type of menu item. Meaningful values are `separator`, `check`, or `radio`. Flex treats all other values, or nodes with no type entry, as normal menu entries. If you use the default data descriptor, data providers must use a `type` XML attribute or object field to specify this characteristic.

Menu-based controls ignore all other object fields or XML attributes, so you can use them for application-specific data.

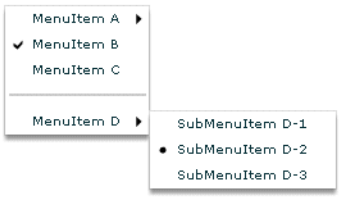
Example: An Array menu data provider

The following example displays a Menu that uses an Array data provider and shows how you define the menu characteristics in the data provider. For an application that specifies an identical menu structure in XML, see "Example: Creating a simple Menu control" on page 429.

```
<?xml version="1.0"?>
<!-- menus/ArrayDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">

    <mx:Script>
        <![CDATA[
            import mx.controls.Menu;

            // Method to create an Array-based menu.
            private function createAndShow():void {
                // The third parameter sets the showRoot property to false.
                // You must set this property in the createMenu method,
                // not later.
                var myMenu:Menu = Menu.createMenu(null, menuData, true);
                myMenu.show(10, 10);
            }

            // The Array data provider
            [Bindable]
            public var menuData:Array = [
                {label: "MenuItem A", children: [
                    {label: "SubMenuItem A-1", enabled: false},
                    {label: "SubMenuItem A-2", type: "normal"}
                    ]},
                {label: "MenuItem B", type: "check", toggled: true},
                {label: "MenuItem C", type: "check", toggled: false},
                {type: "separator"},
                {label: "MenuItem D", children: [
                    {label: "SubMenuItem D-1", type: "radio",
                        groupName: "g1"},
                    {label: "SubMenuItem D-2", type: "radio",
                        groupName: "g1", toggled: true},
                    {label: "SubMenuItem D-3", type: "radio",
                        groupName: "g1"}
                    ]}
                ];
        ]]>
    </mx:Script>

    <!-- Button control to create and open the menu. -->
    <mx:Button x="300" y="10"
        label="Open Menu"
```

```
        click="createAndShow();"/>
</mx:Application>
```

The following image shows the resulting control, with MenuItem D open; notice that check item B and radio item D-2 are selected:



Example: An XML menu data provider with icons

The following example displays a menu control that uses XML as the data provider, and specifies custom icons for the items in the control:

```
<?xml version="1.0"?>
<!-- menus/SimpleMenuControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
            // Import the Menu control.
            import mx.controls.Menu;

            [Bindable]
            [Embed(source="assets/topIcon.jpg")]
            public var myTopIcon:Class;

            [Bindable]
            [Embed(source="assets/radioIcon.jpg")]
            public var myRadioIcon:Class;

            [Bindable]
            [Embed(source="assets/checkIcon.gif")]
            public var myCheckIcon:Class;

            // Create and display the Menu control.
            private function createAndShow():void {
                var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
                myMenu.labelField="@label";

                // Specify the check icon.
                myMenu.setStyle('checkIcon', myCheckIcon);

                // Specify the radio button icon.
                myMenu.setStyle('radioIcon', myRadioIcon);

                // Specify the icon for the topmenu items.
                myMenu.iconField="@icon";
                myMenu.show(10, 10);
            }
        ]]>
    </mx:Script>

    <!-- Define the menu data. -->
    <mx:XML format="e4x" id="myMenuData">
        <root>
            <menuitem label="MenuItem A" icon="myTopIcon">
                <menuitem label="SubMenuItem A-1" enabled="False"/>
                <menuitem label="SubMenuItem A-2"/>
            </menuitem>
            <menuitem label="MenuItem B" type="check" toggled="true"/>
            <menuitem label="MenuItem C" type="check" toggled="false"
                icon="myTopIcon"/>
```

```
            <menuitem type="separator"/>
            <menuitem label="MenuItem D" icon="myTopIcon">
                <menuitem label="SubMenuItem D-1" type="radio"
                    groupName="one"/>
                <menuitem label="SubMenuItem D-2" type="radio"
                    groupName="one" toggled="true"/>
                <menuitem label="SubMenuItem D-3" type="radio"
                    groupName="one"/>
            </menuitem>
        </root>
    </mx:XML>

    <mx:VBox>
        <!-- Define a Button control to open the menu -->
        <mx:Button id="myButton"
            label="Open Menu"
            click="createAndShow();"/>
    </mx:VBox>
</mx:Application>
```

Handling menu-based control events

User interaction with a Menu or menu-based control is event-driven; that is, applications typically handle events generated when the user opens, closes, or selects within a menu, or submenu or rolls over or out of menu items. For detailed information on events and how to use them, see Chapter 5, "Using Events," on page 83.

The Menu and MenuBar controls dispatch an identical set of menu-specific events. Event handling with PopUpMenuButton controls differs from the other two controls, but shares many elements in common with the others.

Handling Menu control events

The Menu control defines the following menu-specific event types, of the MenuEvent class:

change (MenuEvent.CHANGE) Dispatched when a user changes current menu selection by using the keyboard or mouse.

itemClick (MenuEvent.ITEM_CLICK) Dispatched when a user selects an enabled menu item of type `normal`, `check`, or `radio`. This event is not dispatched when a user selects a menu item of type `separator`, a menu item that opens a submenu, or a disabled menu item.

itemRollOut (MenuEvent.ITEM_ROLL_OUT) Dispatched when the mouse pointer rolls off of a Menu item.

itemRollOver (MenuEvent.ITEM_ROLL_OVER) Dispatched when the mouse pointer rolls onto a Menu item.

menuHide (MenuEvent.MENU_HIDE) Dispatched when the entire menu or a submenu closes.

menuShow (MenuEvent.MENU_SHOW) Dispatched when the entire menu or a submenu opens.

The event object passed to the event listener is of type MenuEvent and contains one or more of the following menu-specific properties:

Property	Description
`item`	The item in the data provider for the menu item associated with the event.
`index`	The index of the item in the menu or submenu that contains it.
`label`	The label of the item.
`menu`	A reference to the Menu control where the event occurred.
`menuBar`	The MenuBar control instance that is the parent of the menu, or `undefined` when the menu does not belong to a MenuBar. For more information, see "MenuBar control" on page 431.

To access properties and fields of an object-based menu item, you specify the menu item field name, as follows:

```
ta1.text = event.item.label
```

To access attributes of an E4X XML-based menu item, you specify the menu item attribute name in E4X syntax, as follows:

```
ta1.text = event.item.@label
```

NOTE

If you set an event listener on a submenu of a menu-based control, and the menu data provider's structure changes (for example, an element is removed), the event listener might no longer exist. To ensure that the event listener is available when the data provider structure changes, either listen on events of the menu-based control, not a submenu, or add the event listener each time an event occurs that changes the data provider's structure.

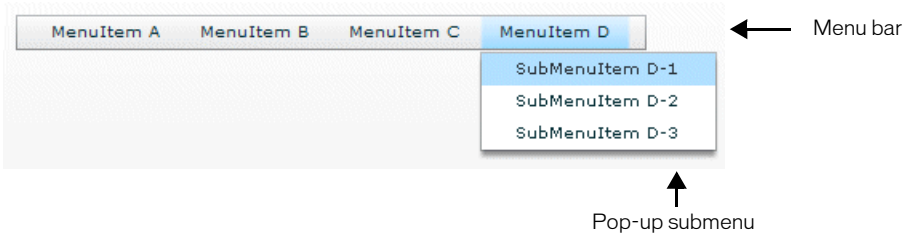
The following example shows a menu with a simple event listener. For a more complex example, see "Example: Using Menu control events" on page 422.

```
<?xml version="1.0"?>
<!-- menus/EventListener.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">

    <mx:Script>
        <![CDATA[
            import mx.controls.Menu;
            import mx.events.MenuEvent;

            // Function to create and show a menu.
            private function createAndShow():void {
                var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
                myMenu.labelField="@label"
                // Add an event listener for the itemClick event.
                myMenu.addEventListener(MenuEvent.ITEM_CLICK,
                    itemClickInfo);
                // Show the menu.
                myMenu.show(225, 10);
            }

            // The event listener for the itemClick event.
            private function itemClickInfo(event:MenuEvent):void {
                ta1.text="event.type: " + event.type;
                ta1.text+="\nevent.index: " + event.index;
                ta1.text+="\nItem label: " + event.item.@label
                ta1.text+="\nItem selected: " + event.item.@toggled;
                ta1.text+= "\nItem type: " + event.item.@type;
            }
        ]]>
    </mx:Script>

    <!-- The XML-based menu data provider. -->
    <mx:XML id="myMenuData">
            <xmlRoot>
            <menuitem label="MenuItem A" >
                <menuitem label="SubMenuItem A-1" enabled="false"/>
                <menuitem label="SubMenuItem A-2"/>
            </menuitem>
            <menuitem label="MenuItem B" type="check" toggled="true"/>
            <menuitem label="MenuItem C" type="check" toggled="false"/>
            <menuitem type="separator"/>
            <menuitem label="MenuItem D" >
                <menuitem label="SubMenuItem D-1" type="radio"
                    groupName="one"/>
                <menuitem label="SubMenuItem D-2" type="radio"
                    groupName="one" toggled="true"/>
```

```
                <menuitem label="SubMenuItem D-3" type="radio"
                    groupName="one"/>
            </menuitem>
            </xmlRoot>
    </mx:XML>

    <!-- Button controls to open the menus. -->
    <mx:Button x="10" y="5"
        label="Open Menu"
        click="createAndShow();"/>
    <!-- Text area to display the event information -->
    <mx:TextArea x="10" y="40"
        width="200" height="100"
        id="ta1"/>
</mx:Application>
```

Handling MenuBar events

The following figure shows a MenuBar control:



For the menu bar, the following events occur:

change (MenuEvent.CHANGE) Dispatched when a user changes current menu bar selection by using the keyboard or mouse. This event is also dispatched when the user changes the current menu selection in a pop-up submenu. When the event occurs on the menu bar, the `menu` property of the MenuEvent object is `null`.

itemRollOut (MenuEvent.ITEM_ROLL_OUT) Dispatched when the mouse pointer rolls off of a menu bar item.

itemRollOver (MenuEvent.ITEM_ROLL_OVER) Dispatched when the mouse pointer rolls onto a menu bar item.

menuHide (MenuEvent.MENU_HIDE) Dispatched when a pop-up submenu closes.

menuShow (MenuEvent.MENU_SHOW) Dispatched when a pop-up submenu opens, or the user selects a menu bar item with no drop-down menu.

NOTE	The MenuBar control does not dispatch the `itemClick` event when you select an item on the menu bar; it only dispatches the `itemClick` event when you select an item on a pop-up submenu.

For each pop-up submenu, the MenuBar dispatches the `change`, `itemClick`, `itemRollOut`, `itemRollOver`, `menuShow`, and `menuHide` events in the same way it does for the Menu control. Handle events triggered by the pop-up menus as you would handle events from Menu controls. For more information, see "Handling Menu control events" on page 415.

The following example handles events for the menu bar and for the pop-up submenus:

```
<?xml version="1.0"?>
<!-- menus/MenuBarEventInfo.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initCollections();">

    <mx:Script>
        <![CDATA[
            import mx.events.MenuEvent;
            import mx.controls.Alert;
            import mx.collections.*;

            [Bindable]
            public var menuBarCollection:XMLListCollection;

            private var menubarXML:XMLList =<>
                <menuitem label="Menu1">
                    <menuitem label="MenuItem 1-A" data="1A"/>
                    <menuitem label="MenuItem 1-B" data="1B"/>
                </menuitem>
                <menuitem label="Menu2">
                    <menuitem label="MenuItem 2-A" data="2A"/>
                    <menuitem label="MenuItem 2-B" data="2B"/>
                </menuitem>
                <menuitem label="Menu3" data="M3"/>
                </>

            // Event handler to initialize the MenuBar control.
            private function initCollections():void {
                menuBarCollection = new XMLListCollection(menubarXML);
            }

            // Event handler for the MenuBar control's change event.
            private function changeHandler(event:MenuEvent):void  {
                // Only open the Alert for a selection in a pop-up submenu.
                // The MenuEvent.menu property is null for a change event
                // dispatched by the menu bar.
                if (event.menu != null) {
                    Alert.show("Label: " + event.item.@label + "\n" +
                        "Data: " + event.item.@data, "Clicked menu item");
                }
            }

            // Event handler for the MenuBar control's itemRollOver event.
            private function rollOverHandler(event:MenuEvent):void {
                rollOverTextArea.text = "type: " + event.type + "\n";
                rollOverTextArea.text += "target menuBarIndex: " +
                    event.index + "\n";
            }
```

```
            // Event handler for the MenuBar control's itemClick event.
            private function itemClickHandler(event:MenuEvent):void {
                itemClickTextArea.text = "type: " + event.type + "\n";
                itemClickTextArea.text += "target menuBarIndex: " +
                    event.index + "\n";
            }
        ]]>
    </mx:Script>

    <mx:Panel title="MenuBar Control Example"
        height="75%" width="75%"
        paddingTop="10" paddingLeft="10">

        <mx:Label
            width="100%"
            color="blue"
            text="Select a menu item."/>
        <mx:MenuBar labelField="@label"
            dataProvider="{menuBarCollection}"
            change="changeHandler(event);"
            itemClick="itemClickHandler(event);"
            itemRollOver="rollOverHandler(event);"/>
        <mx:TextArea id="rollOverTextArea"
            width="200" height="100"/>
        <mx:TextArea id="itemClickTextArea"
            width="200" height="100"/>
    </mx:Panel>
</mx:Application>
```

Example: Using Menu control events

The following example lets you experiment with Menu control events. It lets you display two menus, one with an XML data provider and one with an Array data provider. A TextArea control displays information about each event as a user opens the menus, moves the mouse, and selects menu items. It shows some of the differences in how you handle XML and object-based menus, and indicates some of the types of information that are available about each Menu event.

```
<?xml version="1.0"?>
<!-- menus/ExtendedMenuExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">

    <mx:Script>
        <![CDATA[

        // Import the Menu control and MenuEvent class.
        import mx.controls.Menu;
        import mx.events.MenuEvent;
        //Define a variable for the Menu control.
        private var myMenu:Menu;

        // The event listener that creates menu with an XML data
        // provider and adds event listeners for the menu.
        private function createAndShow():void {
            // Clear the event output display.
            ta1.text="";
            // Don't show the (single) XML root node in the menu.
            myMenu = Menu.createMenu(null, myMenuData, false);
            //You must set the labelField explicitly for XML data providers.
            myMenu.labelField="@label"
            myMenu.addEventListener(MenuEvent.ITEM_CLICK, menuShowInfo);
            myMenu.addEventListener(MenuEvent.MENU_SHOW, menuShowInfo);
            myMenu.addEventListener(MenuEvent.MENU_HIDE, menuShowInfo);
            myMenu.addEventListener(MenuEvent.ITEM_ROLL_OUT, menuShowInfo);
            myMenu.addEventListener(MenuEvent.ITEM_ROLL_OVER,
                menuShowInfo);
            myMenu.show(225, 10);
        }

        // The event listener for the menu events.
        // Retain information on all events for a menu instance.
        private function menuShowInfo(event:MenuEvent):void {
            ta1.text="event.type: " + event.type;
            ta1.text+="\nevent.label: " + event.label;
            // The index value is -1 for menuShow and menuHide events.
            ta1.text+="\nevent.index: " + event.index;
            //The item field is null for show and hide events.
```

```
    if (event.item) {
        ta1.text+="\nItem label: " + event.item.@label
        ta1.text+="\nItem selected: " + event.item.@toggled;
        ta1.text+= "\nItem type: " + event.item.@type;
    }
}

// The event listener that creates an object-based menu
// and adds event listeners for the menu.
private function createAndShow2():void {
    // Show the top (root) level objects in the menu.
    myMenu = Menu.createMenu(null, menuData, true);
    myMenu.addEventListener(MenuEvent.ITEM_CLICK, menuShowInfo2);
    myMenu.addEventListener(MenuEvent.MENU_SHOW, menuShowInfo2);
    // The following line is commented to so you can see the
    // results of an ITEM_CLICK event.
    // (The menu hides immediately after the click.)
    // myMenu.addEventListener(MenuEvent.MENU_HIDE, menuShowInfo2);
    myMenu.addEventListener(MenuEvent.ITEM_ROLL_OVER,
        menuShowInfo2);
    myMenu.addEventListener(MenuEvent.ITEM_ROLL_OUT,
        menuShowInfo2);
    myMenu.show(225, 10);
}

// The event listener for the object-based Menu events.
private function menuShowInfo2(event:MenuEvent):void {
    ta1.text="event.type: " + event.type;
    ta1.text+="\nevent.label: " + event.label;
    // The index value is -1 for menuShow and menuHide events.
    ta1.text+="\nevent.index: " + event.index;
    // The item field is null for show and hide events.
    if (event.item) {
        ta1.text+="\nItem label: " + event.item.label
        ta1.text+="\nItem selected: " + event.item.toggled;
        ta1.text+= "\ntype: " + event.item.type;
    }
}

// The object-based data provider, an Array of objects.
// Its contents is identical to that of the XML data provider.
[Bindable]
public var menuData:Array = [
    {label: "MenuItem A", children: [
        {label: "SubMenuItem A-1", enabled: false},
        {label: "SubMenuItem A-2", type: "normal"}
    ]},
    {label: "MenuItem B", type: "check", toggled: true},
    {label: "MenuItem C", type: "check", toggled: false},
            {type: "separator"},
```

```
            {label: "MenuItem D", children: [
                {label: "SubMenuItem D-1", type: "radio", groupName: "g1"},
                {label: "SubMenuItem D-2", type: "radio", groupName: "g1",
                        toggled: true},
                {label: "SubMenuItem D-3", type: "radio", groupName: "g1"}
            ]}
        ];

    ]]>
    </mx:Script>

    <!-- The XML-based menu data provider.
        The <mx:XML tag requires a single root. -->
    <mx:XML id="myMenuData">
            <xmlRoot>
            <menuitem label="MenuItem A" >
                <menuitem label="SubMenuItem A-1" enabled="false"/>
                <menuitem label="SubMenuItem A-2"/>
            </menuitem>
            <menuitem label="MenuItem B" type="check" toggled="true"/>
            <menuitem label="MenuItem C" type="check" toggled="false"/>
            <menuitem type="separator"/>
            <menuitem label="MenuItem D" >
                <menuitem label="SubMenuItem D-1" type="radio"
                    groupName="one"/>
                <menuitem label="SubMenuItem D-2" type="radio"
                    groupName="one" toggled="true"/>
                <menuitem label="SubMenuItem D-3" type="radio"
                    groupName="one"/>
            </menuitem>
            </xmlRoot>
    </mx:XML>

    <!-- Button controls to open the menus. -->
    <mx:Button x="10" y="5"
        label="Open XML Popup"
        click="createAndShow();"/>
    <mx:Button x="10" y="35"
        label="Open Object Popup"
        click="createAndShow2();"/>
    <!-- Text area to display the event information -->
    <mx:TextArea x="10" y="70"
        width="200" height="300"
        id="ta1"/>
</mx:Application>
```

Handling PopUpMenuButton control events

Because the PopUpMenuButton is a subclass of the PopUpButton control, it supports all of that control's events. When the uer clisks the main button, the PopUpMenuButton control dispatches a `click` (`MouseEvent.CLICK`) event.

When the user clicks the PopUpMenuButton main button, the control dispatches an `itemClick` (MenuEvent.`ITEM_CLICK`) event that contains information about the selected menu item. Therefore, the same `itemClick` event is dispatched when the user clicks the main button or selects the current item from the pop-up menu. Because the same event is dispatched in both cases, clicking on the main button produces the same behavior as clicking on the last selected menuItem, so the main button plays the role of a frequently used menu item.

The following example shows how the PopUpMenuButton generates events and how an application can handle them.

When the user selects an item from the pop-up menu, the following things occur:

- The PopUpMenuButton dispatches an `itemClick` event.
- The application's `itemClickHandler()` event listener function handles the `itemClick` event and displays the information about the event in an Alert control.

When the user clicks the main button, the following things occur:

- The PopUpMenuButton control dispatches a `click` event.
- The PopUpMenuButton control dispatches an `itemClick` event.
- The application's `itemClickHandler()` event listener function handles the `itemClick` event and displays information about the selected Menu item in an Alert control.

- The application's `clickHandler()` event listener function also handles the `MouseEvent.CLICK` event, and displays the Button label in an Alert control.

```
<?xml version="1.0"?>
<!-- menus/PopUpMenuButtonEvents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="600" width="600"
    creationComplete="initData();">

    <mx:Script>
        <![CDATA[
        import mx.events.*;
        import mx.controls.*;

        // Set the Inbox (fourth) item in the menu as the button item.
        private function initData():void {
            Menu(p1.popUp).selectedIndex=3;
        }

        // itemClick event handler, invoked when you select from the menu.
        // Shows the event's label, index properties, and the values of the
        // label and data fields of the data provider entry specified by
        // the event's item property.
        public function itemClickHandler(event:MenuEvent):void {
            Alert.show("itemClick event label: " + event.label
                + "  \nindex: " + event.index
                + "  \nitem.label: " + event.item.label
                + "  \nitem.data: " + event.item.data);
        }

        //Click event handler for the main button.
        public function clickHandler(event:MouseEvent):void {
            Alert.show(" Click Event currentTarget.label: "
                + event.currentTarget.label);
        }

        //The menu data provider
        [Bindable]
        public var menuDP:Array = [
            {label: "Inbox", data: "inbox"},
            {label: "Calendar", data: "calendar"},
            {label: "Sent", data: "sent"},
            {label: "Deleted Items", data: "deleted"},
            {label: "Spam", data: "spam"}
        ];
    ]]></mx:Script>

    <mx:PopUpMenuButton  id="p1"
        showRoot="true"
        dataProvider="{menuDP}"
```

```
        click="clickHandler(event)"
        itemClick="itemClickHandler(event);"
    />
</mx:Application>
```

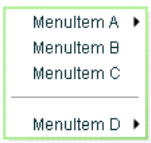
Menu control

The Menu control is a pop-up control that contains a menu of individually selectable choices. You use ActionScript to create a Menu control that pops up in response to a user action, typically as part of an event listener. Because you create a Menu control in response to an event, it does not have an MXML tag; you can create Menu controls in ActionScript only.

For complete reference information, see Menu in *Adobe Flex 2 Language Reference*.

About the Menu Control

The following example shows a Menu control:



In this example, the MenuItem A and MenuItem D items open submenus. Submenus open when the user moves the mouse pointer over the parent item or accesses the parent item by using keyboard keys.

The default location of the Menu control is the upper-left corner of your application, at *x*, *y* coordinates 0,0. You can pass `x` and `y` arguments to the `show()` method to control the position relative to the application.

After a Menu opens, it remains visible until the user selects an enabled menu item, the user selects another component in the application, or a script closes the menu.

To create a *static* menu that stays visible all the time, use the MenuBar control or PopUpMenuButton control. For more information on the MenuBar control, see "MenuBar control" on page 431. For more information on the PopUpMenuButton control, see "PopUpMenuButton control" on page 433.

The Menu control has the following sizing characteristics:

Property	Default value
Default size	The width is determined from the Menu text. The default height is the number of menu rows multiplied by 19 pixels per row (the default row height).

Creating a Menu control

You cannot create a Menu control by using an MXML tag; you must create it in ActionScript.

To create a Menu control:

1. Create an instance of the Menu control by calling the static ActionScript `Menu.createMenu()` method and passing the method an instance of the data provider that contains the information that populates the control as the second parameter; for example:

   ```
   var myMenu:Menu = Menu.createMenu(null, myMenuData);
   ```

   (The first parameter can optionally specify the parent container of the menu.)

   If you do not display the root node of the data provider, for example, if the data provider is an XML document in E4X format, use a third parameter with the value `false`. This parameter sets the menu's `showRoot` property. The following example creates a menu that does not show the data provider root:

   ```
   var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
   ```

   NOTE To hide the root node, you must set the `showRoot` property in the `createMenu` method. Setting the property after you create the menu has no effect.

2. Display the Menu instance by calling the ActionScript `Menu.show()` method; for example:

   ```
   myMenu.show(10, 10);
   ```

NOTE The `show()` method automatically adds the Menu object to the display list, and the `hide()` method automatically removes it from the display list. Clicking outside the menu (or pressing Escape) also hides the Menu object and removes it from the display list.
If you create a Menu object by using the `Menu.createMenu()` method, the Menu is removed from the display list automatically when it is closed. To prevent this default behavior, you can listen for the `menuHide` event and call `preventDefault()` on the event object.
Menus displayed using the `Menu.popUpMenu()` method are not removed automatically; you must call the `PopUpManager.removePopUp()` method on the Menu object.

Example: Creating a simple Menu control

The following example uses the <mx:XML> tag to define the data for the Menu control and a Button control to trigger the event that opens the Menu control:

```
<?xml version="1.0"?>
<!-- menus/SimpleMenuControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
            // Import the Menu control.
            import mx.controls.Menu;

            // Create and display the Menu control.
            private function createAndShow():void {
                var myMenu:Menu = Menu.createMenu(null, myMenuData, false);
                myMenu.labelField="@label";
                myMenu.show(10, 10);
            }
        ]]>
    </mx:Script>

    <!-- Define the menu data. -->
    <mx:XML format="e4x" id="myMenuData">
        <root>
            <menuitem label="MenuItem A" >
                <menuitem label="SubMenuItem A-1" enabled="False"/>
                <menuitem label="SubMenuItem A-2"/>
            </menuitem>
            <menuitem label="MenuItem B" type="check" toggled="true"/>
            <menuitem label="MenuItem C" type="check" toggled="false"/>
            <menuitem type="separator"/>
            <menuitem label="MenuItem D" >
                <menuitem label="SubMenuItem D-1" type="radio"
                    groupName="one"/>
                <menuitem label="SubMenuItem D-2" type="radio"
                    groupName="one" toggled="true"/>
                <menuitem label="SubMenuItem D-3" type="radio"
                    groupName="one"/>
            </menuitem>
        </root>
    </mx:XML>

    <mx:VBox>
        <!-- Define a Button control to open the menu -->
        <mx:Button id="myButton"
            label="Open Menu"
            click="createAndShow();"/>
    </mx:VBox>
```

```
</mx:Application>
```

You can assign any name to node tags in the XML data. In the previous sample, each node is named with the generic `<menuitem>` tag, but you can have used `<node>`, `<subNode>`, `<person>`, `<address>` and so on.

Because this example uses an E4X XML data source, you must specify the label field using the E4X @ attribute specifier syntax, and you tell the control not to show the data provider root node.

Several attributes or fields. such as the `type` attribute, have meaning to the Menu control. For information on how Flex interprets and uses the data provider data, see "Specifying and using menu entry information" on page 409.

Menu control user interaction

You can use the mouse or the keyboard to interact with a Menu control. Clicking selects a menu item and closes the menu, except with the following types of menu items:

Disabled items or separators Rolling over or clicking menu items has no effect and the menu remains visible.

Submenu anchors Rolling over the items activates the submenu; clicking them has no effect; rolling onto any menu item other than one of the submenu items closes the submenu.

When a Menu control has focus, you can use the following keys to control it:

Key	Description
Down Arrow Up Arrow	Moves the selection down and up the rows of the menu. The selection loops at the top or bottom row.
Right Arrow	Opens a submenu, or moves the selection to the next menu in a menu bar.
Left Arrow	Closes a submenu and returns focus to the parent menu (if a parent menu exists), or moves the selection to the previous menu in a menu bar (if the menu bar exists).
Enter	Opens a submenu, has the effect of clicking and releasing the mouse on a row if a submenu does not exist.
Escape	Closes a menu level.

MenuBar control

A MenuBar control displays the top level of a menu as a horizontal bar of menu items, where each item on the bar can pop up a submenu. The MenuBar control interprets the data provider in the same way as the Menu control, and supports the same events as the Menu control. Unlike the Menu control, a MenuBar control is static; that is, it does not function as a pop-up menu, but is always visible in your application. Because the MenuBar is static, you can define it directly in MXML

For complete reference information, see MenuBar in *Adobe Flex 2 Language Reference*. For more information on the Menu control, see "Handling Menu control events" on page 415.

About the MenuBar control

The following example shows a MenuBar control:

The control shows the labels of the top level of the data provider menu. When a user selects a top-level menu item, the MenuBar control opens a submenu. The submenu stays open until the user selects another top-level menu item, selects a submenu item, or clicks outside the MenuBar area.

The MenuBar control has the following sizing characteristics:

Property	Default value
Default size	The width is determined from the menu text, with a minimum value of 27 pixels for the width. The default value for the height is 22 pixels.

Creating a MenuBar control

You define a MenuBar control in MXML by using the `<mx:MenuBar>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block.

You specify the data for the MenuBar control by using the `dataProvider` property. The MenuBar control uses the same types of data providers as does the Menu control. For more information on data providers for Menu and MenuBar controls, see "Defining menu structure and data" on page 408. For more information on hierarchical data providers, see "Using hierarchical data providers" on page 197.

In a simple case for creating a MenuBar control, you might use an <mx:XML> or <mx:XMLList> tag and standard XML node syntax to define the menu data provider. When you used an XML-based data provider, you must keep the following rules in mind:

- With the <mx:XML> tag you must have a single root node, and you set the showRoot property of the MenuBar control to false. (otherwise, your MenuBar would have only the root as a button). With the <mx:XMLList> tag you define a list of XML nodes, and the top level nodes define the bar buttons.
- If your data provider has a label attribute (even if it is called "label"), you must set the MenuBar control's labelField property and use the E4X @ notation for the label; for example:

  ```
  labelField="@label"
  ```

The dataProvider property is the default property of the MenuBar control, so you can define the XML or XMLList object as a direct child of the <mx:MenuBar> tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- menus/MenuBarControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

<!-- Define the menu; dataProvider is the default MenuBar property.
Because this uses an XML data provider, specify the labelField and
showRoot properties. -->
<mx:MenuBar id="myMenuBar" labelField="@label">
<mx:XMLList>
<menuitem label="MenuItem A" >
<menuitem label="SubMenuItem A-1" enabled="False"/>
 <menuitem label="SubMenuItem A-2"/>
</menuitem>
<menuitem label="MenuItem B" type="check" selected="true"/>
<menuitem label="MenuItem C" type="check" selected="false"/>
<menuitem label="MenuItem D" >
<menuitem label="SubMenuItem D-1" type="radio" groupName="one"/>
<menuitem label="SubMenuItem D-2" type="radio" groupName="one"
selected="true"/>
<menuitem label="SubMenuItem D-3" type="radio" groupName="one"/>
</menuitem>
</mx:XMLList>
</mx:MenuBar>
</mx:Application>
```

The top-level nodes in the MenuBar control correspond to the buttons on the bar. Therefore, in this example, the MenuBar control displays the four labels shown in the preceding image.

You can assign any name to node tags in the XML data. In the previous example, each node is named with the generic `<menuitem>` tag, but you can use `<node>`, `<subNode>`, `<person>`, `<address>`, and so on. Several attributes or fields. such as the `type` attribute, have meaning to the MenuBar control. For information on how Flex interprets and uses the data provider data, see "Specifying and using menu entry information" on page 409.

MenuBar control user interaction

The user interaction of the MenuBar is the same as for the Menu control, with the following difference: When the MenuBar control is has the focus, the left arrow opens the previous menu. If the current menu bar item has a closed pop-up menu, the right arrow opens the current menu; if the pop-up menu is open, the right arrow opens the next menu. (The behavior wraps around the ends of the MenuBar control.)

For more information, see "Menu control user interaction" on page 430.

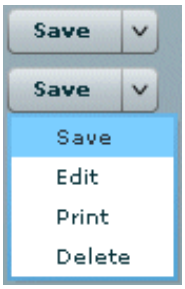
PopUpMenuButton control

The PopUpMenuButton is a PopUpButton control whose secondary button pops up a Menu control. When the user selects an item from the pop-up menu, the main button of the PopUpButton changes to show the icon and label of the selected menu item. Unlike the Menu and MenuBar controls, the PopUpMenuButton supports only a single-level menu.

For complete reference information, see PopUpMenuButton in *Adobe Flex 2 Language Reference*. For more information on the Menu control, see "Handling Menu control events" on page 415. For more information on PopUpButton controls, see "PopUpButton control" on page 273.

About the PopUpMenuButton control

The following example shows a PopUpMenuButton control before and after clicking the secondary pop-up button:



The PopUpMenuButton work as follows:

- When you click the smaller button, which by default displays a **v** icon, the control displays a pop-up menu below the button.
- When you select an item from the pop-up menu, the main PopUpMenuButton button label changes to show the selected item's label and the PopUpMenuButton control dispatches a `MenuEvent.CHANGE` event.
- When you click the main button, the PopUpMenuButton control dispatches a `MenuEvent.CHANGE` event and a `MouseEvent.ITEM_CLICK` event.

For information on handling PopUpMenuButton events, see "Handling PopUpMenuButton control events" on page 425

The PopUpMenuButton control lets users change the function of the main button by selecting items from the pop-up menu. The most recently selected item becomes the main button item.

This behavior is useful for buttons when there are a number of user actions, users tend to select the same option frequently, and the application developer cannot assume which option should be the default. Text editors often use such controls in their control bar for options, such as spacing, for which a user is likely to have a preferred setting, but the developer cannot determine it in advance. Microsoft Word, for example, uses such controls for specifying line spacing, borders, and text and highlight color.

You can use the PopUpButton control to create pop-up menu buttons with behaviors that differ from those of the PopUpMenuButton; for example, buttons that do not change the default action of the main button when the user selects a menu item. For more information, see "PopUpButton control" on page 273.

The PopUpMenuButton control has the following sizing characteristics:

Property	Default value
Default size	Sufficient to accommodate the label and any icon on the main button, and the icon on the pop-up button. The control does not reserve space for the menu.
Minimum size	0
Maximum size	10000 by 10000

Creating a PopUpMenuButton control

You define a PopUpMenuButton control in MXML by using the `<mx:PopUpMenuButton>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block.

You specify the data for the PopUpMenuButton control by using the `dataProvider` property. For information on valid data providers, including their structure and contents, see "Defining menu structure and data" on page 408.

By default, the initially selected item is the first item in the pop-up menu dataProvider, and the default main button label is the item's label, as determined by the `labelField` or `labelFunction` property. To set the initial main button label to a specific item's label and functionality, write a listener for the PopUpMenuButton control's `creationComplete` event that sets the `selectedIndex` property of the Menu subcontrol, as follows:

```
Menu(MyPopUpControl.popUp).selectedIndex=2;
```

You must cast the PopUpMenuButton control's `popUp` property to a Menu because the property type is IUIComponent, not Menu.

You can also use the `label` property of the PopUpMenuButton control to set the main button label, as described in "Using the label property" on page 437.

When a popped up menu closes, it loses its selection and related properties.

NOTE You must use the PopUpMenuButton's `creationComplete` event, not the `initialize` event to set the main button label from the data provider.

Example: Creating a PopUpMenuButton control

The following example creates a PopUpMenuButton control using an E4X XML data provider.

```
<?xml version="1.0"?>
<!-- menus/PopUpMenuButtonControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Menu

            // The initData function sets the initial value of the button
           // label by setting the Menu subcontrol's selectedIndex property.
            // You must cast the popUp property to a Menu.
            private function initData():void {
                Menu(pb2.popUp).selectedIndex=2;
            }
        ]]>
    </mx:Script>

    <mx:XML format="e4x" id="dp2">
        <root>
            <editItem label="Cut"/>
            <editItem label="Copy"/>
            <editItem label="Paste"/>
            <separator type="separator"/>
            <editItem label="Delete"/>
        </root>
    </mx:XML>

    <mx:PopUpMenuButton id="pb2"
        dataProvider="{dp2}"
        labelField="@label"
        showRoot="false"
        creationComplete="initData();"/>
</mx:Application>
```

Because this example uses an E4X XML data source, you must specify the label field using the E4X @ attribute specifier syntax, and you must tell the control not to show the data provider root node.

Using the label property

The `label` property of the PopUpMenuButton control specifies the contents of the label on the main button, and overrides any label from the pop-up menu that is determined by the `labelField` or `labelFunction` property. The `label` property is useful for creating a main button label with fixed and a variable parts; for example, a mail "Send to:" button where only the destination text is controlled by the pop-up menu, so the main button could say "Send to: Inbox" or "Send to: Trash" based on the selection from a menu that lists "Menu" and "Trash."

To use a dynamic `label` property, use a PopUpMenuButton control `change` event listener to set the label based on the event's `label` property, as in the following example:

```
<?xml version="1.0"?>
<!-- menus/PopUpMenuButtonLabel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="600" width="600">

    <mx:Script>
        <![CDATA[
            import mx.events.MenuEvent;

            public function itemClickHandler(event:MenuEvent):void {
                event.currentTarget.label= "Send to: " + event.label;
            }

            [Bindable]
            public var menuData:Array = [
                {label: "Inbox", data: "inbox"},
                {label: "Calendar", data: "calendar"},
                {label: "Sent", data: "sent"},
                {label: "Deleted Items", data: "deleted"},
                {label: "Spam", data: "spam"}
            ];
        ]]>
    </mx:Script>

    <mx:PopUpMenuButton id="p1"
        showRoot="true"
        dataProvider="{menuData}"
        label="Send to: Inbox"
        itemClick="itemClickHandler(event);"/>
</mx:Application>
```

PopUpMenuButton user interaction

The user interaction of the PopUpMenuButton control main button and secondary button is the same as for the PopUpButton control. The user interaction with the pop-up menu is the same as for the Menu control. For more information on the PopUpButton user interaction, see "User interaction" on page 276. For more information on Menu control user interaction, see "Menu control user interaction" on page 430.

CHAPTER 12

Using Data-Driven Controls

12

Several Adobe Flex controls take input from a data provider, an object that contains data. For example, a Tree control reads data from a data provider to define the structure of the tree and any associated data assigned to each tree node.

This topic describes several of the controls that use a data provider, focusing on controls that let you visualize complex data. It includes examples of different ways to populate these controls by using a data provider. The following topics also provide information on data providers and controls that use data providers:

- Chapter 7, "Using Data Providers and Collections," on page 161 contains details on data providers and how to use collections as data providers.
- Chapter 11, "Using Menu-Based Controls," on page 407 contains information on using Menu, MenuBar, and PopUpMenuButton controls.
- Chapter 16, "Using Navigator Containers," on page 627 contains information on navigator containers, such as TabNavigator and Accordion that use data providers to populate their structures.
- Chapter 53, "Introduction to Charts," on page 1573 contains information on using chart controls.

Contents

List control

The List control displays a vertical list of items. Its functionality is very similar to that of the `SELECT` form element in HTML. It often contains a vertical scroll bar that lets users access the items in the list. An optional horizontal scroll bar lets users view items when the full width of the list items is unlikely to fit. The user can select one or more items from the list.

NOTE	The HorizontalList, TileList, DataGrid, Menu, and Tree controls are derived from the List control or its immediate parent, the ListBase class. As a result, much of the information for the List control applies to these controls.

For complete reference information, see List in *Adobe Flex 2 Language Reference*.

The following image shows a List control:

List control sizing

The List control has the following default sizing properties:

Property	Default value
Default size	Wide enough to fit the widest label in the first seven visible items (or all visible items in the list, if there are less than seven), and seven rows high, where each row is 20 pixels high.
Minimum size	0
Maximum size	5000 by 5000

If you specify `horizontalScrollPolicy="on"`, the default width of a List control does not change; it is still large enough to display the widest visible label. If you set `horizontalScrollPolicy="on"`, and specify a List control pixel width, you can use the `measureWidthOfItems()` method to ensure that the scroll bar rightmost position corresponds to the right edge of the content, as the following example shows. Notice that the additional 5 pixels ensures that the rightmost character of the text displays properly.

```
<mx:List width="200" id="li2" horizontalScrollPolicy="on"
  maxHorizontalScrollPosition="{li2.measureWidthOfItems() - li2.width +
  5}">
```

The preceding line ensures that the rightmost position of the scroll bar puts the end of the longest measured list item near the right edge of the List control. Using this technique, however, can reduce application efficiency, so you might consider using explicit sizes instead.

Lists, and all subclasses of the ListBase class determine their sizes when a style changes or the data provider changes.

If you set a `width` property that is less than the width of the longest label and specify the `horizontalScrollPolicy="off"`, labels that exceed the control width are clipped.

Creating a List control

You use the `<mx:List>` tag to define a List control. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The List control uses a list-based data provider. For more information, see "About data providers" on page 162.

You specify the data for the List control by using the `dataProvider` property of the control. However, because `dataProvider` is the List control's default property, you do not have to specify a `<mx:dataProvider>` child tag of the `<mx:List>` tag. In the simplest case for creating a static List control, you need only put `<mx:String>` tags in the control body, because Flex also automatically interprets the multiple tags as an Array of Strings, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:List>
        <mx:dataProvider>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
            <mx:String>AR</mx:String>
        </mx:dataProvider>
    </mx:List>
</mx:Application>
```

The index of items in the List control is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items. The value of the item is its label text.

You typically use events to handle user interaction with a List control. The following example code adds a handler for a `change` event to the List control. Flex broadcasts a `mx.ListEvent.CHANGE` event when the value of the control changes due to user interaction.

```
<?xml version="1.0"?>
<!-- dpcontrols/ListChangeEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            public function changeEvt(event:Event):void {
               forChange.text=event.currentTarget.selectedItem.label + " " +
                 event.currentTarget.selectedIndex;
            }
        ]]>
    </mx:Script>

    <mx:List width="35" change="changeEvt(event)">
        <mx:Object label="AL" data="Montgomery"/>
        <mx:Object label="AK" data="Juneau"/>
        <mx:Object label="AR" data="Little Rock"/>
    </mx:List>
    <mx:TextArea id="forChange" width="150"/>
</mx:Application>
```

In this example, you use two properties of the List control, `selectedItem` and `selectedIndex`, in the event handler. Every `change` event updates the TextArea control with the label of the selected item and the item's index in the control.

The `target` property of the object passed to the event handler contains a reference to the List control. You can reference any control property using the event's `currentTarget` property. The `currentTarget.selectedItem` field contains a copy of the selected item. If you populate the List control with an Array of Strings, the `currentTarget.selectedItem` field contains a String. If you populate it with an Array of Objects, the `currentTarget.selectedItem` field contains the Object that corresponds to the selected item, so, in this case, `currentTarget.selectedItem.label` refers to the selected item's label field.

Using a label function

You can pass a label function to the List control to provide logic that determines the text that appears in the control. The label function must have the following signature:

```
labelFunction(item:Object):String
```

The *item* parameter passed in by the Label control contains the list item object. The function must return the string to display in the List control.

NOTE

Most subclasses of ListBase also take a `labelFunction` property with the signature described above. For the DataGrid and DataGridColumn controls, the method signature is: *`labelFunction(item:Object, dataField:DataGridColumn):String`*
where *`item`* contains the DataGrid item object, and *`dataField`* specifies the DataGrid column.

The following example uses a function to combine the values of the `label` and `data` fields for each item for display in the List control:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListLabelFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
    <mx:Script><![CDATA[
        public function myLabelFunc(item:Object):String {
            return item.data + ", " + item.label;
        }
    ]]></mx:Script>

    <mx:ArrayCollection id="myDP">
        <mx:source>
            <mx:Object label="AL" data="Montgomery"/>
            <mx:Object label="AK" data="Juneau"/>
            <mx:Object label="AR" data="Little Rock"/>
        </mx:source>
    </mx:ArrayCollection>

    <mx:List dataProvider="{myDP}" labelFunction="myLabelFunc"/>
</mx:Application>
```

This example creates the following List control:

NOTE

This example uses an ArrayCollection object as the data provider. You should use collections as the data providers if the data can change dynamically. For more information, see Chapter 7, "Using Data Providers and Collections," on page 161.

Displaying DataTips

DataTips are similar to ToolTips, but display text when the mouse pointer hovers over a row in a List control. Text in a List control that is longer than the control width is clipped on the right side (or requires scrolling, if the control has scroll bars). DataTips can solve that problem by displaying all of the text, including the clipped text, when the mouse pointer hovers over a cell. If you enable data tips, they only appear for fields where the data is clipped. To display DataTips, set the `showDataTips` property of a List control to `true`.

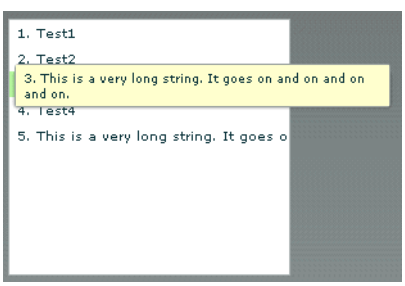
NOTE To use DataTips with a DataGrid control, you must set the `showDataTips` property on the individual DataGridColumns of the DataGrid.

The default behavior of the `showDataTips` property is to display the label text. However, you can use the `dataTipField` and `dataTipFunction` properties to determine what is displayed in the DataTip. The `dataTipField` property behaves like the `labelField` property; it specifies the name of the field in the data provider to use as the DataTip for cells in the column. The `dataTipFunction` property behaves like the `labelFunction` property; it specifies the DataTip string to display for list items.

The following example sets the `showDataTips` property for a List control:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  showDataTips="true"/>
```

This example creates the following List control:



Displaying ScrollTips

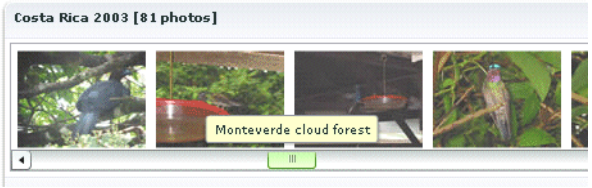
You use ScrollTips to give users context about where they are in a list as they scroll through the list. The tips appear only when you scroll; they don't appear if you only hover the mouse over the scroll bar. ScrollTips are useful when live scrolling is disabled (the `liveScrolling` property is false) so scrolling does not occur until you release the scroll thumb. The default value of the `showScrollTips` property is `false`.

The default behavior of the `showScrollTips` property is to display the index number of the top visible item. You can use the `scrollTipFunction` property to determine what is displayed in the ScrollTip. The `scrollTipFunction` property behaves like the `labelFunction` property; it specifies the ScrollTip string to display for list items. You should avoid going to the server to fill in a ScrollTip.

The following example sets the `showScrollTips` and `scrollTipFunction` properties of a HorizontalList control. The `scrollTipFunction` property specifies a function that gets the value of the `description` property of the current list item.

```
<mx:HorizontalList id="list" dataProvider="{album.photo}" width="100%"
  itemRenderer="Thumbnail" columnWidth="108" height="100"
  selectionColor="#FFCC00" liveScrolling="false" showScrollTips="true"
  scrollTipFunction="scrollTipFunc"
  change="currentPhoto=album.photo[list.selectedIndex]"/>
```

This code produces the following HorizontalList control:



Vertically aligning text in List control rows

You can use the `verticalAlign` style to vertically align text at the top, middle, or bottom of a List row. The default value is `top`. You can also specify a value of `middle` or `bottom`.

The following example sets the `verticalAlign` property for a List control to `bottom`:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  verticalAlign="bottom"/>
```

Setting variable row height and wrapping List text

You can use the `variableRowHeight` property to make the height of List control rows variable based on their content. The default value is `false`. If you set the `variableRowHeight` property to `true`, the `rowHeight` property is ignored and the `rowCount` property is read-only.

The following example sets the `variableRowHeight` property for a List control to `true`:

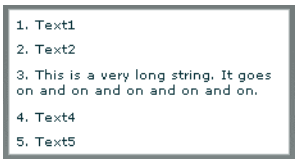
```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  variableRowHeight="true"/>
```

You can use the `wordWrap` property in combination with the `variableRowHeight` property to wrap text to multiple lines when it exceeds the width of a List row.

The following example sets the `wordWrap` and `variableRowHeight` properties to `true`:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  variableRowHeight="true" wordWrap="true"/>
```

This code produces the following List control:



Using a custom item renderer

A item renderer is the object that displays a List control's data items. The simplest way to use a custom item renderer is to specify an MXML component as the value of the `itemRenderer` property. When you use an MXML component as a item renderer, it can contain multiple levels of containers and controls. You can also use an ActionScript class as a custom item renderer. For detailed information on custom item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 851

The following example sets the `itemRenderer` property to an MXML component named FancyCellRenderer. It also sets the `variableRowHeight` property to `true` because the MXML component exceeds the default row height:

```
<mx:List id="myList1" dataProvider="{myDP}" width="220" height="200"
  itemRenderer="FancyItemRenderer" variableRowHeight="true"/>
```

Specifying an icon to the List control

You can specify an icon to display with each List item, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListIcon.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:Script>
      <![CDATA[
         [Embed(source="assets/radioIcon.jpg")]
         public var iconSymbol1:Class;
         [Embed(source="assets/topIcon.jpg")]
         public var iconSymbol2:Class;
      ]]>
   </mx:Script>

   <mx:List iconField="myIcon">
      <mx:dataProvider>
         <mx:Array>
            <mx:Object label="AL" data="Montgomery" myIcon="iconSymbol1"/>
            <mx:Object label="AK" data="Juneau" myIcon="iconSymbol2"/>
            <mx:Object label="AR" data="Little Rock" myIcon="iconSymbol1"/>
         </mx:Array>
      </mx:dataProvider>
   </mx:List>
</mx:Application>
```

In this example, you use the `iconField` property to specify the field of each item containing the icon. You use the `Embed` metadata to import the icons, and then reference them in the List control definition.

You can also use the `iconFunction` property to specify a function that determines the icon, similar to the way that you can use the `labelFunction` property to specify a function that determines the label text. The icon function must have the following signature:

```
iconFunction(item:Object):Class
```

The *`item`* parameter passed in by the Label control contains the list item object. The function must return the icon class to display in the List control.

The following example shows a List control that uses the `iconFunction` property to determine the icon to display for each item in the list:

```
<?xml version="1.0"?>
<!-- dpcontrols/ListIconFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
   <mx:Script>
      <![CDATA[
         // Embed icons.
         [Embed(source="assets/radioIcon.jpg")]
         public var pavementSymbol:Class;
         [Embed(source="assets/topIcon.jpg")]
         public var normalSymbol:Class;

         // Define data provider.
         private var myDP: Array;
         private function initList():void {
            myDP = [
               {Artist:'Pavement', Album:'Slanted and Enchanted',
Price:11.99},
               {Artist:'Pavarotti', Album:'Twilight', Price:11.99},
               {Artist:'Other', Album:'Other', Price:5.99}];

            list1.dataProvider = myDP;
         }

         // Determine icon based on artist. Pavement gets a special icon.
         private function myiconfunction(item:Object):Class{
            var type:String = item.Artist;
            if (type == "Pavement") {
               return pavementSymbol;
            }
            return normalSymbol;
         }
      ]]>
   </mx:Script>

   <mx:VBox >
      <mx:List id="list1" initialize="initList()" labelField="Artist"
         iconFunction="myiconfunction" />
   </mx:VBox>
</mx:Application>
```

Alternating row colors in a List control

You can use the `alternatingItemColors` style property to specify an Array that defines the color of each row in the List control. The Array must contain two or more colors. After using all the entries in the Array, the List control repeats the color scheme.

The following example defines an Array with two entries, #66FFFF for light blue and #33CCCC for a blue-gray. Therefore, the rows of the List control alternate between these two colors. If you specify a three-color array, the rows alternate among the three colors, and so on.

```
<mx:List alternatingItemColors="[#66FFFF, #33CCCC]".../ >
```

List control user interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items. (You must set the `allowMultipleSelection` property to `true` to allow multiple selecton.)

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the List control broadcasts this event when the mouse button is released.

If you set the `allowDragSelection` property to `true`, the control scrolls up or down when the user presses the mouse button over the one or more rows, holds the mouse button down, drags the mouse outside the control, and then moves the mouse up and down.

A List control shows the number of records that fit in the display. Paging down through the data displayed by a 10-line List control shows records 0-10, 9-18, 18-27, and so on, with one line overlapping from one page to the next.

The List control has the following keyboard navigation features:

Key	Action
Up Arrow	Moves selection up one item.
Down Arrow	Moves selection down one item.
Page Up	Moves selection up one page.
Page Down	Moves selection down one page.
Home	Moves selection to the top of the list.
End	Moves selection to the bottom of the list.
Alphanumeric keys	Jumps to the next item with a label that begins with the character typed.
Control	Toggle key. Allows for multiple (noncontiguous) selection and deselection. Works with key presses, click selection, and drag selection.
Shift	Contiguous selection key. Allows for contiguous selections. Works with key presses, click selection, and drag selection.

HorizontalList control

The HorizontalList control displays a horizontal list of items. The HorizontalList control is particularly useful in combination with a custom item renderer for displaying a list of images and other data. For more information about custom item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

For complete reference information, see HorizontalList in *Adobe Flex 2 Language Reference*.

About HorizontaList controls

The contents of a HorizontalList control can look very similar to the contents of an HBox container in which a Repeater object repeats components. However, performance of a HorizontalList control can be better than the combination of an HBox container and a Repeater object because the HorizontalList control only instantiates the objects that fit in its display area. Scrolling in a HorizontalList can be slower than it is when using a Repeater object. For more information about the Repeater object, see Chapter 26, "Dynamically Repeating Controls and Containers," on page 995.

The HorizontalList control always displays items from left to right. The control usually contains a horizontal scroll bar, which lets users access all items in the list. An optional vertical scroll bar lets users view items when the full height of the list items is unlikely to fit. The user can select one or more items from the list, depending on the value of the `allowMultipleSelection` property.

The following image shows a HorizontalList control:

Product images courtesy of Lavish

The HorizontalList control has the following default sizing properties:

Property	Default value
Default size	Four columns, with the dimensions determined by the cell dimensions.
Minimum size	0
Maximum size	5000 by 5000

For complete reference information, see HorizontalList in *Adobe Flex 2 Language Reference*.

Creating a HorizontalList control

You use the `<mx:HorizontalList>` tag to define a HorizontalList control. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The HorizontalList control shares many properties and methods with the List control; see "List control" on page 440 for information on how to use several of these shared properties. The HorizontalList control uses a list-based data provider. For more information, see "About data providers" on page 162.

You specify the data for a HorizontalList control using the `dataProvider` property of the `<mx:HorizontalList>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/HListDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="450">
   <mx:Script>
      <![CDATA[
         import mx.collections.*;
         import mx.controls.Image;

         private var catalog:ArrayCollection;
         private static var cat:Array = [
            "assets/usbfan.jpg", "assets/usbwatch.jpg",
            "assets/007camera.jpg", "assets/radiowatch.jpg"
         ];

         // Initialize the HorizontalList control by setting its
dataProvider
         // property to an ArrayCollection containing the items parameter.
         private function initCatalog(items:Array):void
         {
            catalog = new ArrayCollection(items);
            myList.dataProvider = catalog;
         }
      ]]>
   </mx:Script>

   <!-- A four-column HorizontalList.
       The itemRenderer is a Flex Image control.
       When the control is created, pass the cat array to the
       initialization routine. -->
   <mx:HorizontalList id="myList" columnWidth="100" rowHeight="100"
      columnCount="4" itemRenderer="mx.controls.Image"
      creationComplete="initCatalog(cat)"/>
</mx:Application>
```

In this example, you use the creationComplete event to populate the data provider with an ArrayCollection of image files, and the `itemRenderer` property to specify the Image control as the item renderer. (Note that you use the full package name of the control in the assignment because the code does not import the mx.controls package.) The HorizontalList control then displays the four images specified by the data provider.

The following examples are based on the Flex Explorer application included in Flex 2.

The HorizontalListDemo.mxml example displays a catalog of product images in a HorizontalList control. The item renderer for the HorizontalList control is an MXML component named Thumbnail.

```
<?xml version="1.0"?>
<!-- dpcontrols/HorizontalListDemo.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
   backgroundAlpha="0" creationComplete="srv.send()">

   <mx:Script>
      <![CDATA[
         import mx.utils.ArrayUtil;
      ]]>
   </mx:Script>


   <mx:HTTPService id="srv" url="assets/catalog.xml" useProxy="false"/>

   <mx:ArrayCollection id="catalogAC"

source="{mx.utils.ArrayUtil.toArray(srv.lastResult.catalog.product)}"/>

   <mx:HorizontalList dataProvider="{catalogAC}"
      width="100%" itemRenderer="Thumbnail"/>
</mx:Application>
```

The following example shows the Thumbnail.mxml MXML component that is used as the item renderer in the product catalog application. In this example, you define the item renderer to contain three controls: an Image control and two Label controls. These controls examine the `data` object passed to the item renderer to determine the content to display. For more information about custom item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

```
<?xml version="1.0"?>
<!-- dpcontrols/Thumbnail.mxml -->
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml"
      width="165" height="120" verticalAlign="middle" verticalGap="0"
      verticalScrollPolicy="off">
   <mx:CurrencyFormatter id="cf"/>
   <mx:Image id="img" height="100" width="50" source="../{data.image}"/>
   <mx:VBox width="100%" paddingTop="0" horizontalGap="4">
      <mx:Label text="{data.name}" fontWeight="bold"/>
      <mx:Label text="{cf.format(data.price)}" fontWeight="bold"/>
   </mx:VBox>
</mx:HBox>
```

HorizontalList control user interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items.

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the HorizontalList control broadcasts this event when the mouse button is released. When the user drags the mouse over the items and then outside the control, the control scrolls up or down.

A HorizontalList control shows the number of records that fit in the display. Paging through a four list shows records 0-4, 5-8, and so on, with no overlap from one page to the next.

Keyboard navigation

The HorizontalList control has the following keyboard navigation features:

Key	Action
Page Up	Moves selection to the left one page.
Left Arrow	Moves selection to the left one item.
Down Arrow	Moves selection right one item.
Page Down	Moves selection to the right one page.
Home	Moves selection to the beginning of the list.

Key	Action
End	Moves selection to the end of the list.
Control	Toggle key. Allows for multiple (noncontiguous) selection and deselection when the `allowMultipleSelection` property is set to `true`. Works with key presses, click selection, and drag selection.
Shift	Contiguous selection key. Allows for contiguous selections when `allowMultipleSelection` is set to `true`. Works with key presses, click selection, and drag selection.

TileList control

The TileList control displays a tiled list of items. The items are tiled in vertical columns or horizontal rows. The TileList control is particularly useful in combination with a custom item renderer for displaying a list of images and other data. The default item renderer for the TileList control is TileListItemRenderer, which, by default, displays text of the data provider's label field and any icon. For more information about custom item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

For complete reference information, see TileList in *Adobe Flex 2 Language Reference*.

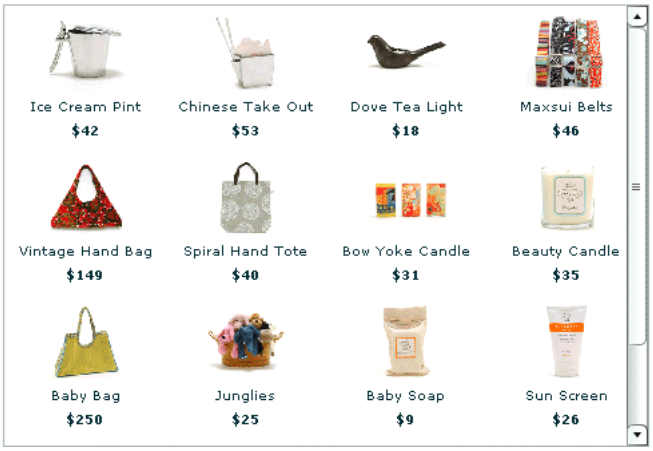
About the TileList control

The contents of a TileList control can look very similar to the contents of a Tile container in which a Repeater object repeats components. However, performance of a TileList control can be better than the combination of a Tile container and a Repeater object because the TileList control only instantiates the objects that fit in its display area. Scrolling in a TileList can be slower than it is when using a Repeater object. For more information about the Repeater object, see Chapter 26, "Dynamically Repeating Controls and Containers," on page 995.

The TileList control displays a number of items laid out in equally sized tiles. It often contains a scroll bar on one of its axes to access all items in the list depending on the direction orientation of the list. The user can select one or more items from the list depending on the value of the `allowMultipleSelection` property.

The TileList control lays out its children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The `direction` property determines the primary direction of the layout. The valid values for the `direction` property are and `horizontal` (default) and `vertical` for a layout. In a `horizontal` layout the tiles are filled in row by row with each row filling the available space in the control. If there are more tiles than fit in the display area, the horizontal control has a vertical scroll. In a `vertical` layout, the tiles are filled in column by column in the available vertical space, and the control may have a horizontal scroll bar.

The following image shows a TileList control:



The TileList control has the following default sizing properties:

Property	Default value
Default size	Four columns and four rows. Using the default item renderer, each cell is 50 by 50 pixels, and the total size is 200 by 200 pixels.
Minimum size	0
Maximum size	5000 by 5000

For complete reference information, see TileList in *Adobe Flex 2 Language Reference*.

Creating a TileList control

You use the `<mx:TileList>` tag to define a TileList control. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The TileList control shares many properties and methods with the List control; see "List control" on page 440 for information on how to use several of these shared properties. The TileList control uses a list-based data provider. For more information, see "About data providers" on page 162.

You specify the data for a TileList control using the `dataProvider` property of the `<mx:TileList>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/TileListDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
   initialize="initData();" >
   <mx:Script>
   <![CDATA[
      import mx.controls.Button;
      import mx.collections.*;
      private var listArray:Array=[
         {label: "item0", data: 0},{label: "item1", data: 1},
         {label: "item2", data: 2},{label: "item3", data: 3},
         {label: "item4", data: 4},{label: "item5", data: 5},
         {label: "item6", data: 6},{label: "item7", data: 7},
         {label: "item8", data: 8}];
      [Bindable]
      public var TileListdp:ArrayCollection;

      private function initData():void {
         TileListdp = new ArrayCollection(listArray);
      }
   ]]>
   </mx:Script>

   <mx:TileList dataProvider="{TileListdp}"
      itemRenderer="mx.controls.Button"/>
</mx:Application>
```

In this example, you populate the data provider with an ArrayCollection that contains an Array of strings defining labels and data values. You then use the `itemRenderer` property to specify a Button control as the item renderer. The Button controls display the data provider label values. The TileList control displays nine Button controls with the specified labels.

The next example is the TileListDemo.mxml file from the Flex Explorer sample application; this example shows the MXML code for a catalog application that displays a set of product images in a TileList control. The item renderer for the TileList control is an MXML component named Thumbnail. In this example, you define the item renderer to contain three controls: an Image control and two Label controls. These controls examine the `data` object passed to the item renderer to determine the content to display. For more information about custom item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

```
<?xml version="1.0"?>
<!-- dpcontrols/TileListDemo.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
   backgroundAlpha="0" creationComplete="srv.send()">

   <mx:HTTPService id="srv" url="assets/catalog.xml" useProxy="false"/>

   <mx:TileList dataProvider="{srv.lastResult.catalog.product}"
      height="100%" width="100%" itemRenderer="Thumbnail"
      rowHeight="130" columnWidth="175"/>
</mx:Application>
```

The Thumbnail.mxml MXML component that is used as the item renderer in the product catalog application is identical to the one in "HorizontalList control" on page 450. For more information about custom item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

TileList control user interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items.

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the TileList control broadcasts this event when the mouse button is released. When the user drags the mouse over the items and then outside the control, the control scrolls up or down.

Keyboard navigation

The TileList control has the following keyboard navigation features:

Key	Action
Up Arrow	Moves selection up one item. If the control direction is `vertical`, and the current item is at the top of a column, moves to the last item in the previous column; motion stops at the first item in the first column.
Down Arrow	Moves selection down one item.If the control direction is `vertical`, and the current item is at the bottom of a column, moves to the first item in the next column; motion stops at the last item in the last column.
Right Arrow	Moves selection to the right one item. If the control direction is `horizontal`, and the current item is at the end of a row, moves to the first item in the next row; motion stops at the last item in the last column.
Left Arrow	Moves selection to the left one item. If the control direction is `horizontal`, and the current item is at the beginning of a row, moves to the last item in the previous row; motion stops at the first item in the first row.
Page Up	Moves selection up one page. For a single-page control, moves the selection to the beginning of the list.
Page Down	Moves selection down one page. For a single-page control, moves the selection to the end of the list.
Home	Moves selection to the beginning of the list.
End	Moves selection to the end of the list.
Control	Toggle key. Allows for multiple (noncontiguous) selection and deselection when `allowMultipleSelection` is set to `true`. Works with key presses, click selection, and drag selection.
Shift	Contiguous selection key. Allows for contiguous selections when `allowMultipleSelection` is set to `true`. Works with key presses, click selection, and drag selection.

ComboBox control

The ComboBox control is a drop-down list from which the user can select a single value. Its functionality it is very similar to that of the SELECT form element in HTML.

For complete reference information, see ComboBox in *Adobe Flex 2 Language Reference*.

About the ComboBox control

The following image shows a ComboBox control:

In its editable state, the user can type text directly into the top of the list, or select one of the preset values from the list. In its noneditable state, as the user types a letter, the drop-down list opens and scrolls to the value that most closely matches the one being entered; matching is only performed on the first letter that the user types.

If the drop-down list hits the lower boundary of the application, it opens upward. If a list item is too long to fit in the horizontal display area, it is truncated to fit. If there are too many items to display in the drop-down list, a vertical scroll bar appears.

The ComboBox control has the following default sizing properties:

Property	Default value
Default size	Wide enough to accommodate the longest entry in the drop-down list in the display area of the main control, plus the drop-down button. When the drop-down list is not visible, the default height is based on the label text size. The default drop-down list height is five rows, or the number of entries in the drop-down list, whichever is smaller. The default height of each entry in the drop-down list is 22 pixels.
Minimum size	0
Maximum size	5000 by 5000.
`dropdownWidth`	The width of the ComboBox control.
`rowCount`	5

Creating a ComboBox control

You use the `<mx:ComboBox>` tag to define a ComboBox control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The ComboBox control uses a list-based data provider. For more information, see "About data providers" on page 162.

You specify the data for the ComboBox control using the `dataProvider` property of the `<mx:ComboBox>` tag. The data provider should be an Array, or a class that implements the ICollectionView or IList interface; often it is an ArrayCollection. For more information on data providers and collections see Chapter 7, "Using Data Providers and Collections," on page 161.

In a simple case for creating a ComboBox control, you specify the property using an `<mx:dataProvider>` child tag, and use an `<mx:ArrayCollection>` tag to define the entries as an ArrayCollection whose source is an Array of Strings, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:ComboBox>
      <mx:ArrayCollection>
         <mx:String>AK</mx:String>
         <mx:String>AL</mx:String>
         <mx:String>AR</mx:String>
      </mx:ArrayCollection>
   </mx:ComboBox>
</mx:Application>
```

This example shows how you can take advantages of MXML defaults. You do not have to use an `<mx:dataProvider>` tag, because `dataProvider` is the default property of the ComboBox control. Similarly, you do not have to use an `<mx:source>` tag inside the `<mx:ArrayCollection>` tag because `source` is the default property of the ArrayCollection class. Finally, you do not have to specify an `<mx:Array>` tag for the source array.

The data provider can also contain objects with multiple fields, as in the following example:

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxMultiple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:ComboBox>
      <mx:ArrayCollection>
         <mx:Object label="AL" data="Montgomery"/>
         <mx:Object label="AK" data="Juneau"/>
         <mx:Object label="AR" data="Little Rock"/>
      </mx:ArrayCollection>
   </mx:ComboBox>
</mx:Application>
```

If the data source is an array of strings, as in the first example, the ComboBox displays strings as the items in the drop-down list. If the data source consists of objects, the ComboBox, by default, uses the contents of the label field. You can, however, override this behavior, as described in "Specifying ComboBox labels" on page 463.

The index of items in the ComboBox control is zero-based, which means that values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of items. The value of the item is its label text.

Using events with ComboBox controls

You typically use events to handle user interaction with a ComboBox control.

The ComboBox control broadcasts a `change` event (flash.events.Event class with a `type` property value of flash.events.Event.CHANGE) when the value of the control's `selectedIndex` or `selectedItem` property changes due to the following user actions:

- If the user closes the drop-down list using a mouse click, Enter key, or Control+Up key, and the selected item is different from the previously selected item.
- If the drop-down list is currently closed, and the user presses the Up, Down, Page Up, or Page Down key to select a new item.
- If the ComboBox control is editable, and the user types into the control, Flex broadcasts a `change` event each time the text field of the control changes.

The ComboBox control broadcasts an mx.events.DropdownEvent with a type mx.events.DropdownEvent.OPEN (`open`) and mx.events.DropdownEvent.CLOSE (close) when the ComboBox control opens and closes. For detailed information on these and other ComboBox events, see ComboBox in *Adobe Flex 2 Language Reference*.

The following example displays information from ComboBox events:

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
   <mx:Script>
      <![CDATA[
         import flash.events.Event;
         import mx.events.DropdownEvent;

         // Display the type of event for open and close events.
         private function dropEvt(event:DropdownEvent):void {
            forChange.text+=event.type + "\n";
         }

         // Display a selected item's label field and index for change
events.
         private function changeEvt(event:Event):void {
            forChange.text+=event.currentTarget.selectedItem.label + " " +
               event.currentTarget.selectedIndex + "\n";
         }
      ]]>
   </mx:Script>

   <mx:ComboBox open="dropEvt(event)" close="dropEvt(event)"
         change="changeEvt(event)" >
      <mx:ArrayCollection>
         <mx:Object label="AL" data="Montgomery"/>
         <mx:Object label="AK" data="Juneau"/>
         <mx:Object label="AR" data="Little Rock"/>
      </mx:ArrayCollection>
   </mx:ComboBox>
   <mx:TextArea id="forChange" width="150" height="100%"/>
</mx:Application>
```

If you populate the ComboBox control with an Array of Strings, the `currentTarget.selectedItem` field contains a String. If you populate it with an Array of Objects, the `currentTarget.selectedItem` field contains the Object that corresponds to the selected item, so, in this case, `currentTarget.selectedItem.label` refers to the selected item object's label field.

In this example, you use two properties of the ComboBox control, `selectedItem` and `selectedIndex`, in the event handlers. Every `change` event updates the TextArea control with the label of the selected item and the item's index in the control, and every `open` or `close` event appends the event type.

Specifying ComboBox labels

If the ComboBox data source is an array of strings, the control displays the string for each item. If the data source is contains Objects, by default, the ComboBox control expects each object to contain a property named `label` that defines the text that appears in the ComboBox control for the item. If each Object does not contain a `label` property, you can use the `labelField` property of the ComboBox control to specify the property name, as the following example shows:

```
<mx:ComboBox open="dropEvt(event)" close="dropEvt(event)"
     change="changeEvt(event)" labelField="state">
  <mx:ArrayCollection>
    <mx:Object state="AL" capital="Montgomery"/>
    <mx:Object state="AK" capital="Juneau"/>
    <mx:Object state="AR" capital="Little Rock"/>
  </mx:ArrayCollection>
</mx:ComboBox>
```

To make the event handler in the section "Using events with ComboBox controls" on page 461 display the state ID and capital, you would modify the `change` event handler to use a property named `state`, as the following example shows:

```
private function changeEvt(event) {
  forChange.text=event.currentTarget.selectedItem.state + " " +
  event.currentTarget.selectedItem.capital + " " +
  event.currenttarget.selectedIndex;
}
```

You can also specify the ComboBox labels using a label function, as described in Using a label function on page 442.

Populating a ComboBox control using variables and models

Flex lets you populate the data provider of a ComboBox control from an ActionScript variable definition or from a Flex data model. Each element of the data provider must contain a string label, and can contain one or more fields with additional data. The following example populates two ComboBox controls; one from an ArrayCollection variable that it populates directly from an Array, the other from an ArrayCollection that it populates from an array of items in an `<MX:Model>` tag.

```
<?xml version="1.0"?>
<!-- dpcontrols/ComboBoxVariables.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
   initialize="initData();">

   <mx:Script>
      <![CDATA[
         import mx.collections.*
         private var COLOR_ARRAY:Array=
            [{label:"Red", data:"#FF0000"},
            {label:"Green", data:"#00FF00"},
            {label:"Blue", data:"#0000FF"}];
         // Declare an ArrayCollection variable for the colors.
         // Make it Bindable so it can be used in bind
         // expressions ({colorAC}).
         [Bindable]
         public var colorAC:ArrayCollection;

         // Initialize colorAC ArrayCollection variable from the Array.
         // Use an initialize event handler to initialize data variables
         // that do not rely on components, so that the initial values are
         // available when the controls that use them are constructed.
         //See the mx:ArrayCollection tag, below, for a second way to
         //initialize an ArrayCollection.
         private function initData():void {
            colorAC=new ArrayCollection(COLOR_ARRAY);
         }
      ]]>
   </mx:Script>
   <!-- This example shows two different ways to
        structure a Model. -->
   <mx:Model id="myDP">
      <obj>
         <item label="AL" data="Montgomery"/>
         <item>
            <label>AK</label>
            <data>Juneau</data>
         </item>
         <item>
            <label>AR</label>
```

```
            <data>Little Rock</data>
          </item>
       </obj>
    </mx:Model>

    <!-- Create a stateAC ArrayCollection that uses as its source an Array of
          the item elements from the myDP model.
         This technique and the declaration and initialization code used for
          the colorAC variable are alternative methods of creating and
          initializing the ArrayCollection. -->
    <mx:ArrayCollection id="stateAC" source="{myDP.item}"/>

    <mx:ComboBox dataProvider="{colorAC}"/>
    <mx:ComboBox dataProvider="{stateAC}"/>
</mx:Application>
```

This example uses a simple model. However, you can populate the model from an external data source or define a custom data model class in ActionScript. For more information on using data models, see Chapter 39, "Storing Data," on page 1269.

You can use remote data providers to supply data to your ComboBox control. For example, when a web service operation returns an Array of strings, you can use the following format to display each string as a row of a ComboBox control:

```
<mx:ArrayCollection id="resultAC"
  source="mx.utils.ArrayUtil.toArray(service.operation.lastResult);"
<mx:ComboBox dataProvider="{resultAC}" />
```

For more information on using remote data providers, see "Using remote data providers" on page 213.

ComboBox control user interaction

An ArrayCollection control can be noneditable or editable, as specified by the Boolean `editable` property. In a noneditable ComboBox control, a user can make a single selection from a drop-down list. In an editable ComboBox control, the portion button of the control is a text field that the user can enter text directly into or can populate by selecting an item from the drop-down list. When the user makes a selection in the ComboBox control list, the label of the selection is copied to the text field at the top of the ComboBox control.

When a ComboBox control (and not the drop-down box) has focus and is editable, all keystrokes go to the text field and are handled according to the rules of the TextInput control (see "TextInput control" on page 393), with the exception of the Control+Down key combination, which opens the drop-down list. When the drop-down list is open, you can use the Up and Down keys to navigate in the list and the Enter key to select an item from the list.

When a ComboBox control has focus and is noneditable, alphanumeric keystrokes move the selection up and down the data provider to the next item with the same first character and display the label in the text field. If the drop-down list is open, the visible selection moves to the selected item.

You can also use the following keys to control a noneditable ComboBox control when the drop-down list is not open:

Key	Description
Control+Down	Opens the drop-down list and gives it focus.
Down	Moves the selection down one item.
End	Moves the selection to the bottom of the collection.
Home	Moves the selection to the top of the collection.
Page Down	Displays the item that would be at the end bottom of the drop-down list. If the current selection is a multiple of the `rowCount` value, displays the item that `rowCount` -1 down the list, or the last item. If the current selection is the last item in the data provider, does nothing,.
Page Up	Displays the item that would be at the top of the drop-down. If the current selection is a multiple of the `rowCount` value, displays the item that `rowCount` -1 up the list, or the first item. If the current selection is the first item in the data provider, does nothing,
Up	Moves the selection up one item.

When the drop-down list of a non-editable ComboBox control has focus, alphanumeric keystrokes move the selection up and down the drop-down list to the next item with the same first character. You can also use the following keys to control a drop-down list when it is open:

Key	Description
Control+Up	Closes the drop-down list and returns focus to the ComboBox control.
Down	Moves the selection down one item.
End	Moves the selection to the bottom of the collection.
Enter	Closes the drop-down list and returns focus to the ComboBox control.
Escape	Closes the drop-down list and returns focus to the ComboBox control.
Home	Moves the selection to the top of the collection.
Page Down	Moves to the bottom of the visible list. If the current selection is at the bottom of the list, moves the current selection to the top of the displayed list and displays the next `rowCount`-1 items, if any. If there current selection is the last item in the data provider, does nothing,.

Key	Description
Page Up	Moves to the top of the visible list. If the current selection is at the top of the list, moves the current selection to the bottom of the displayed list and displays the previous `rowCount`-1 items, if any. If the current selection is the first item in the data provider, does nothing,.
Shift+Tab	Closes the drop-down list and moves the focus to the previous object in the DisplayList.
Tab	Closes the drop-down list and moves the focus to the next object in the DisplayList.
Up	Moves the selection up one item.

DataGrid control

The DataGrid control is a list that can display more than one column of data. It is a formatted table of data that lets you set editable table cells, and is the foundation of many data-driven applications.

This topic describes how to create and use DataGrid controls, including how to sort the data. It does not cover information on the following topics, which are often important for creating advanced data grid controls:

- How to format the information in each DataGrid cell and control how users enter data in the cells; for information on these topics, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.
- How to drag objects to and from the data grid; for information on this topic, see Chapter 29, "Using the Drag and Drop Manager," on page 1081.

For complete reference information, see DataGrid in *Adobe Flex 2 Language Reference*.

About the DataGrid control

The DataGrid control provides the following features:

- Resizable, sortable, and customizable column layouts, including hidable columns
- Optional customizable column and row headers, including optionally wrapping header text
- Columns that the user can resize and reorder at run time
- Selection events
- Ability to use a custom item renderer for any column
- Support for paging through data

- Locked rows and columns that do not scroll

The following image shows a DataGrid control:

Artist	Album	Price
Pavement	Slanted and Enchanted	11.99
Pavement	Crooked Rain, Crooked Rain	10.99
Pavement	Wowee Zowee	12.99
Pavement	Brighten the Corners	11.99
Pavement	Terror Twilight	11.99
Other	Other	5.99

Rows are responsible for rendering items. Each row is laid out vertically below the previous one. Columns are responsible for maintaining the state of each visual column; columns control width, color, and size.

The DataGrid control has the following default sizing properties:

Property	Default value
Default size	If the columns are empty, the default width is 300 pixels. If the columns contain information but define no explicit widths, the default width is 100 pixels per column. The DataGrid width is sized to fit the width of all columns, if possible. The default number of displayed rows, including the header is 7, and each row, by default, is 20 pixels high.
Minimum size	0
Maximum size	5000 by 5000

Creating a DataGrid control

You use the `<mx:DataGrid>` tag to define a DataGrid control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The DataGrid control uses a list-based data provider. For more information, see "About data providers" on page 162.

You specify the data for the DataGrid control using the `dataProvider` property. You can specify data in several different ways. In the simplest case for creating a DataGrid control, you use the `<mx:dataProvider>` property subtag with `<mx:ArrayCollection>`, and `<mx:Object>` tags to define the entries as an ArrayCollection of Objects. Each Object defines a row of the DataGrid control, and properties of the Object define the column entries for the row, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:DataGrid>
      <mx:ArrayCollection>
         <mx:Object>
            <mx:Artist>Pavement</mx:Artist>
            <mx:Price>11.99</mx:Price>
            <mx:Album>Slanted and Enchanted</mx:Album>
         </mx:Object>
         <mx:Object>
            <mx:Artist>Pavement</mx:Artist>
            <mx:Album>Brighten the Corners</mx:Album>
            <mx:Price>11.99</mx:Price>
         </mx:Object>
      </mx:ArrayCollection>
   </mx:DataGrid>
</mx:Application>
```

This example shows how you can take advantages of MXML defaults. You do not have to use an `<mx:dataProvider>` tag, because `dataProvider` is the default property of the DataGrid control. Similarly, you do not have to use an `<mx:source>` tag inside the `<mx:ArrayCollection>` tag because `source` is the default property of the ArrayCollection class. Finally, you do not have to specify an `<mx:Array>` tag for the source array.

You can also define the objects using properties directly in the Object tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSimpleAttributes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:DataGrid>
      <mx:ArrayCollection>
         <mx:Object Artist="Pavement"
            Album="Slanted and Enchanted" Price="11.99" />
         <mx:Object Artist="Pavement"
            Album="Brighten the Corners" Price="11.99" />
      </mx:ArrayCollection>
   </mx:DataGrid>
</mx:Application>
```

The column names displayed in the DataGrid control are the property names of the Array Objects. By default, the columns are in alphabetical order by the property names. Different Objects can define their properties in differing orders. If an Array Object omits a property, the DataGrid control displays an empty cell in that row.

Specifying columns

Each column in a DataGrid control is represented by a DataGridColumn object. You use the `columns` property of the DataGrid control and the `<mx:DataGridColumn>` tag to select the DataGrid columns, specify the order in which to display them, and set additional properties. You can also use the DataGridColumn class `visible` property to hide and redisplay columns, as described in "Hiding and displaying columns" on page 471.

For complete reference information for the `<mx:DataGridColumn>` tag, see DataGridColumn in *Adobe Flex 2 Language Reference*.

You specify an Array element to the `<mx:columns>` child tag of the `<mx:DataGrid>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSpecifyColumns.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
   <mx:DataGrid>
      <mx:ArrayCollection>
         <mx:Object Artist="Pavement" Price="11.99"
            Album="Slanted and Enchanted" />
         <mx:Object Artist="Pavement"
            Album="Brighten the Corners" Price="11.99" />
      </mx:ArrayCollection>
      <mx:columns>
         <mx:DataGridColumn dataField="Album" />
         <mx:DataGridColumn dataField="Price" />
      </mx:columns>
   </mx:DataGrid>
</mx:Application>
```

In this example, you only display the Album and Price columns in the DataGrid control. You can reorder the columns as well, as the following example shows:

```
<mx:columns>
  <mx:DataGridColumn dataField="Price" />
  <mx:DataGridColumn dataField="Album" />
</mx:columns>
```

In this example, you specify that the Price column is the first column in the DataGrid control, and that the Album column is the second.

You can also use the <mx:DataGridColumn> tag to set other options. The following example uses the headerText property to set the name of the column to a value different than the default name of Album, and uses the width property to set the album name column wide enough to display the full album names:

```
<mx:columns>
  <mx:DataGridColumn dataField="Album" width="200" />
  <mx:DataGridColumn dataField="Price" headerText="List Price" />
</mx:columns>
```

Hiding and displaying columns

If you might display a column at some times, but not at others, you can specify the DataGridColumn class visible property to hide or show the column. The following example lets you hide or show the album price by clicking a button:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridVisibleColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
   <mx:DataGrid id="myDG" width="350">
      <mx:dataProvider>
         <mx:ArrayCollection>
            <mx:source>
               <mx:Object Artist="Pavement" Price="11.99"
                  Album="Slanted and Enchanted" />
               <mx:Object Artist="Pavement"
                  Album="Brighten the Corners" Price="11.99" />
            </mx:source>
         </mx:ArrayCollection>
      </mx:dataProvider>
      <mx:columns>
         <mx:DataGridColumn dataField="Artist" />
         <mx:DataGridColumn dataField="Album" />
         <mx:DataGridColumn id="price" dataField="Price" visible="false"/>
      </mx:columns>
   </mx:DataGrid>

   <!-- The column id property specifies the column to show.-->
   <mx:Button label="Toggle Price Column"
      click="price.visible = !price.visible;" />
</mx:Application>
```

Passing data to a DataGrid control

Flex lets you populate a DataGrid control from an ActionScript variable definition or from a Flex data model. The following example populates a DataGrid control by using a variable:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridPassData.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
   initialize="initData()">
   <mx:Script>
   <![CDATA[
      import mx.collections.*;
      private var DGArray:Array = [
         {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
         {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}];

      [Bindable]
      public var initDG:ArrayCollection;
      //Initialize initDG ArrayCollection variable from the Array.
      //You can use this technique to convert an HTTPService,
      //WebService, or RemoteObject result to ArrayCollection.
      public function initData():void {
         initDG=new ArrayCollection(DGArray);
      }
   ]]>
   </mx:Script>

   <mx:DataGrid id="myGrid" width="350" height="200"
      dataProvider="{initDG}" >
      <mx:columns>
         <mx:DataGridColumn dataField="Album" />
         <mx:DataGridColumn dataField="Price" />
      </mx:columns>
   </mx:DataGrid>
</mx:Application>
```

In this example, you bind the variable `initDG` to the `<mx:dataProvider>` property. You can still specify a column definition event when using data binding. For a description of using a model as a data provider, see "Populating a ComboBox control using variables and models" on page 464.

Handling events in a DataGrid control

The DataGrid control and the DataGridEvent class define several event types that let you respond to user interaction. For example, Flex broadcasts mx.events.ListEvent class event with a `type` property value of `mx.events.ListEvent.ITEM_CLICK` ("itemClick") when a user clicks an item in a DataGrid control. You can handle this event as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridEvents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:Script>
   <![CDATA[
      import mx.events.ListEvent;
      private function itemClickEvent(event:ListEvent):void {
         clickColumn.text=String(event.columnIndex);
         clickRow.text=String(event.rowIndex);
         eventType.text=event.type;
      }
   ]]>
   </mx:Script>
   <mx:DataGrid id="myGrid" width="350" height="150"
         itemClick="itemClickEvent(event);">
      <mx:ArrayCollection>
         <mx:Object Artist="Pavement" Price="11.99"
            Album="Slanted and Enchanted" />
         <mx:Object Artist="Pavement" Album="Brighten the Corners"
            Price="11.99" />
      </mx:ArrayCollection>
   </mx:DataGrid>

   <mx:TextArea id="clickColumn" />
   <mx:TextArea id="clickRow" />
   <mx:TextArea id="eventType" />
</mx:Application>
```

In this example, you use the event handler to display the column index, row index, and event type in three TextArea controls.

The index of columns in the DataGrid control is zero-based, meaning values are 0, 1, 2, ... , *n* - 1, where *n* is the total number of columns. Row items are also indexed starting at 0. Therefore, if you select the first item in the second row, this example displays 0 in the first Text Area control for the column index, and 1 in the second TextArea control for the item index in the column.

To access the selected item in the event handler, you can use the `currentTarget` property of the event object, and the `selectedItem` property of the DataGrid control, as the following code shows:

```
var selectedArtist:String=event.currentTarget.selectedItem.Artist;
```

The `currentTarget` property of the object passed to the event handler contains a reference to the DataGrid control. You can reference any control property using `currentTarget` followed by a period and the property name. The `currentTarget.selectedItem` field contains the selected item.

Sorting data in DataGrid controls

The DataGrid control supports displaying sorted data in two ways:

- By default, the control displays data in the sorted order of its underlying data provider collection. Therefore you can use the collection Sort and SortField classes to control the order of the rows.
- By default, users can sort the display by clicking the column headers. Clicking the column header initially sorts the display in descending order of the entries in the selected column, and clicking the header again reverses the sort order. You can disable sorting an entire DataGrid Control or individual columns.

For detailed information on using the Sort and SortField classes, see "Sorting and filtering data for viewing" on page 177.

Determining the initial DataGrid sort order

To specify the initial DataGrid sort order, you sort the data provider. While a number of approaches can work, the following technique takes best advantage of the built in features of Flex collections:

- Use an object that implements the ICollectionView interface, such as an ArrayCollection, in the `dataProvider` property of your DataGrid. Specify a Sort object in the data provider object's the `sort` field
- Use the Sort object to control the order of the rows in the dataProvider object.

For an example that sets an initial, multicolumn sort on a DataGrid, see "Example: Sorting a DataGrid on multiple columns" on page 476.

Controlling user sorting of DataGrid displays

Three DataGrid and DataGridColumn properties control how users can sort the order of a data

- The DataGrid `sortableColumns` property is a global switch that enables user sorting of the DataGrid display by clicking column headings. The default this property is `true`.
- The DataGridColumn class `sortable` property specifies whether users can sort an individual column. The default this property is `true`.
- The DataGridColumn class `sortCompareFunction` property lets you specify a custom comparison function. This property sets the `compare` property of the default SortField class object that the DataGrid uses to sort the grid when users click the headers. It lets you specify the function that compares two objects and determines which would be higher in the sort order, without requiring you to explicitly create a Sort object on your data provider. For detailed information on the comparison function signature and behavior, see sortCompareFunction in *Adobe Flex 2 Language Reference*.

By default, the DataGrid class uses its own sort code to control how the data gets sorted when the user clicks a column. To override this behavior, you create a `headerRelease` event handler to handle the DataGridEvent class event that is generated when the user clicks the column header. This event handler must do the following:

1. Use the event object's `columnIndex` property to determine the clicked column.
2. Establish a Sort object with a set of SortField objects based on the clicked column and any other rules that you need to control the sorting order.
3. Apply the Sort object to the data provider ICollectionView.
4. Call the DataGridEvent class event object's `preventDefault()` method to prevent the DataGrid from doing a default column sort.

NOTE

If you specify a `labelFunction` property, you must also specify a `sortCompareFunction` function. The Computed Columns example in Flex Explorer shows this use.

The following example shows how to use the headerRelease event handler to do multi-column sorting when a user clicks a DataGrid column header.

Example: Sorting a DataGrid on multiple columns

The following example shows how you can use a collection with a Sort object to determine an initial multi-column sort and to control how the columns sort when you click the headers. The data grid is initially sorted by in-stock status first, artist second, and album name, third. If you click any heading, that column becomes the primary sort criterion, the previous primary criterion becomes the second criterion, and the previous secondary criterion becomes the third criterion.

```
<?xml version="1.0"?>
<!-- dpcontrols/DataGridSort.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
      initialize="initDP();" width="550" height="400">

   <mx:Script>
      <![CDATA[
         import mx.events.DataGridEvent;
         import mx.collections.*;

         // Declare storage variables and initialize the simple variables.
         // The data provider collection.
         private var myDPColl:ArrayCollection;
         // The Sort object used to sort the collection.
         [Bindable]
         private var sortA:Sort;
         // The sort fields used to determine the sort.
          private var sortByInStock:SortField;
          private var sortByArtist:SortField;
          private var sortByAlbum:SortField;
          private var sortByPrice:SortField;
         // The data source that populates the collection.
         private var myDP:Array = [
            {Artist:'Pavement', Album:'Slanted and Enchanted',
               Price:11.99, InStock: true},
            {Artist:'Pavement', Album:'Crooked Rain, Crooked Rain',
               Price:10.99, InStock: false},
            {Artist:'Pavement', Album:'Wowee Zowee',
               Price:12.99, InStock: true},
            {Artist:'Asphalt', Album:'Brighten the Corners',
               Price:11.99, InStock: false},
            {Artist:'Asphalt', Album:'Terror Twilight',
               Price:11.99, InStock: true},
            {Artist:'Asphalt', Album:'Buildings Meet the Sky',
               Price:14.99, InStock: true},
            {Artist:'Other', Album:'Other', Price:5.99, InStock: true}
         ];

         //Initialize the DataGrid control with sorted data.
         private function initDP():void {
```

```
    //Create an ArrayCollection backed by the myDP array of data.
    myDPColl = new ArrayCollection(myDP);
    //Create a Sort object to sort the ArrrayCollection.
    sortA = new Sort();
    //Initialize SortField objects for all valid sort fields:
    // A true second parameter specifies a case-insensitive sort.
    // A true third parameter specifies descending sort order.
    // A true fourth parameter specifies a numeric sort.
     sortByInStock = new SortField("InStock", true, true);
     sortByArtist = new SortField("Artist", true);
     sortByAlbum = new SortField("Album", true);
     sortByPrice = new SortField("Price", true, false, true);
    // Sort the grid using the InStock, Artist, and Album fields.
    sortA.fields=[sortByInStock, sortByArtist, sortByAlbum];
    myDPColl.sort=sortA;
    // Refresh the collection view to show the sort.
    myDPColl.refresh();
    // Initial display of sort fields
    tSort0.text = "First Sort Field: InStock";
    tSort1.text = "Second Sort Field: Artist";
    tSort2.text = "Third Sort Field: Album";

    // Set the ArrayCollection as the DataGrid data provider.
    myGrid.dataProvider=myDPColl;
    // Set the DataGrid row count to the array length,
    // plus one for the header.
    myGrid.rowCount=myDPColl.length +1;
}

// Re-sort the DataGrid control when the user clicks a header.
private function headRelEvt(event:DataGridEvent):void {
    // The new third priority was the old second priority.
    sortA.fields[2] = sortA.fields[1];
    tSort2.text = "Third Sort Field: " + sortA.fields[2].name;
    // The new second priority was the old first priority.
    sortA.fields[1] = sortA.fields[0];
    tSort1.text = "Second Sort Field: " + sortA.fields[1].name;
    // The clicked column determines the new first priority.
    if (event.columnIndex==0) {
        sortA.fields[0] = sortByArtist;
    } else if (event.columnIndex==1) {
        sortA.fields[0] = sortByAlbum;
    } else if (event.columnIndex==2) {
        sortA.fields[0] = sortByPrice;
    } else {
        sortA.fields[0] = sortByInStock;}
    tSort0.text = "First Sort Field: " + sortA.fields[0].name;
    // Apply the updated sort fields and re-sort.
    myDPColl.sort=sortA;
    // Refresh the collection to show the sort in the grid.
```

```
            myDPColl.refresh();
            // Prevent the DataGrid from doing a default column sort.
            event.preventDefault();
         }
      ]]>
   </mx:Script>

   <!-- The Data Grid control.
         By default the grid and its columns can be sorted by clicking.
         The headerRelease event handler overrides the default sort
         behavior. -->
   <mx:DataGrid id="myGrid" width="100%"
headerRelease="headRelEvt(event);">
      <mx:columns>
            <mx:DataGridColumn minWidth="120" dataField="Artist" />
            <mx:DataGridColumn minWidth="200" dataField="Album" />
            <mx:DataGridColumn width="75" dataField="Price" />
            <mx:DataGridColumn width="75" dataField="InStock"
               headerText="In Stock"/>
      </mx:columns>
   </mx:DataGrid>
   <mx:VBox>
      <mx:Label id="tSort0" text="First Sort Field: "/>
      <mx:Label id="tSort1" text="Second Sort Field: "/>
      <mx:Label id="tSort2" text="Third Sort Field: "/>
   </mx:VBox>
</mx:Application>
```

DataGrid control user interaction

The DataGrid control responds to mouse and keyboard activity. The response to a mouse click or key press depends on whether a cell is editable. A cell is editable when the `editable` properties of the DataGrid control and the DataGridColumn containing the cell are both `true`. Clicking within an editable cell directs focus to that cell. Clicking a noneditable cell has no effect on the focus.

Users can modify the DataGrid control appearance in the following ways:

- If the value of the `sortableColumns` property is `true`, the default value, clicking within a column header causes the DataGrid control to be sorted based on the column's cell values.
- If the value of the `draggableColumns` property is `true`, the default value, clicking and holding the mouse button within a column header, dragging horizontally, and releasing the mouse button moves the column to new location.
- If the value of the `resizableColumns` property is `true`, the default value, clicking in the area between columns permits column resizing.

Keyboard navigation

The DataGrid control has the following keyboard navigation features:

Key	Action
Enter Return Shift+Enter	When a cell is in editing state, commits change, and moves editing to the cell on the same column, next row down or up, depending on whether Shift is pressed.
Tab	Moves focus to the next editable cell, traversing the cells in row order. If at the end of the last row, advances to the next element in the parent container that can receive focus.
Shift+Tab	Moves focus to the previous editable cell. If at the beginning of a row, advances to the end of the previous row. If at the beginning of the first row, advances to the previous element in the parent container that can receive focus.
Up Arrow Home Page Up	If editing a cell, shifts the cursor to the beginning of the cell's text. If the cell is not editable, moves selection up one item.
Down Arrow End Page Down	If editing a cell, shifts the cursor to the end of the cell's text. If the cell is not editable, moves selection down one item.
Control	Toggle key. If you set the DataGrid control `allowMultipleSelection` property to `true`, allows for multiple (noncontiguous) selection and deselection. Works with key presses, click selection, and drag selection.
Shift	Contiguous select key. If you set the DataGrid control `allowMultipleSelection` property to `true`, allows for contiguous selections. Works with key presses, click selection, and drag selection.

Tree control

The Tree control lets a user view hierarchical data arranged as an expandable tree.

For complete reference information, see Tree in *Adobe Flex 2 Language Reference*. For information on hierarchical data providers, see "Using hierarchical data providers" on page 197.

This topic describes how to create and use Tree controls. It does not cover information on the following topics, which are often important for using advanced Tree controls:

- How to format the information in each Tree node and control how users enter data in the nodes; for information on these topics, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

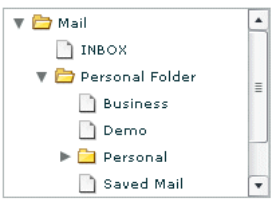
- How to drag objects to and from the Tree control; for information on this topic, see Chapter 29, "Using the Drag and Drop Manager," on page 1081.

About Tree Controls

A Tree control is a hierarchical structure of *branch* and *leaf nodes*. Each item in a tree is called a node and can be either a leaf or a branch. A branch node can contain leaf or branch nodes, or can be empty (have no children). A leaf node is an end point in the tree.

By default, a leaf is represented by a text label beside a file icon and a branch is represented by a text label beside a folder icon with a disclosure triangle that a user can open to expose children.

The following image shows a Tree control:



The Tree control has the following default sizing properties:

Property	Default value
Default size	wide enough to accommodate the icon, label, and expansion triangle, if any, of the widest node in the first 7 displayed (uncollapsed) rows, and seven rows high, where each row is 20 pixels in height. If a scroll bar is required, the width of the scroll bar is not included in the width calculations.
Minimum size	0
Maximum size	5000 by 5000

Creating a Tree control

You define a Tree control in MXML using the `<mx:Tree>` tag. The Tree control is derived from the List control and takes all of the properties and methods of the List control. For more information about using the List control, see "List control" on page 440. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

The Tree control normally gets its data from a hierarchical data provider, such as XML. If the Tree represents dynamically changing data, you use an object that implements the ICollectionView interface, such as ArrayCollection or XMLListCollection.

The Tree control uses a data descriptor to parse and manipulate the data provider contents. By default, the Tree control uses a DefaultDataDescriptor class descriptor, but you can create your own class and specify it in the Menu control's `dataDescriptor` property.

The DefaultDataDescriptor class supports the following types of data:

XML A string containing valid XML text, or any of the following objects containing valid E4X format XML data: `<mx:XML>` or `<mx:XMLList>` compile-time tag, or an XML or XMLList object.

Other objects An array of items, or an object that contains an array of items, where a node's children are contained in an item named `children`.

Collections An object that implements the ICollectionView interface (such as the ArrayCollection or XMLListCollection classes) and whose data source conforms to the structure specified in either of the previous bullets. The DefaultDataDescriptor class includes code to handle collections efficiently. Always use a collection as the data provider if the data in the menu changes dynamically; otherwise the Tree control might display obsolete data.

The DefaultDataDescriptor class also supports using an `<mx:Model>` tag as a data provider for a menu, but all leaf nodes must have the name `children`; As a general rule, it is a better programming practice to use the `<mx:XML>` or `<mx:XMLList>` tags when you need a Tree data provider that uses binding.

For more information on hierarchical objects and data descriptors, including a detailed description of the formats supported by the DefaultDataDescriptor, see "Data descriptors and hierarchical data provider structure" on page 197.

The following code contains a single Tree control that defines the tree shown in the image in "Tree control" on page 479. This uses an XMLListCollection wrapper around an `<mx:XMLList>` tag. By using an XMLListCollection, you can modify the underlying XML data provider by changing the contents of the `MailBox` XMLListCollection, and the Tree control will represent the changes to the data. This example also does not use the <mx:dataProvider> tag because `dataProvider` is the default property of the Tree control.

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:Tree id="tree1" labelField="@label" showRoot="true" width="160">
      <mx:XMLListCollection id="MailBox">
         <mx:XMLList>
            <folder label="Mail">
               <folder label="INBOX"/>
               <folder label="Personal Folder">
                  <Pfolder label="Business" />
                  <Pfolder label="Demo" />
                     <Pfolder label="Personal" isBranch="true" />
                     <Pfolder label="Saved Mail" />
                  </folder>
                  <folder label="Sent" />
                  <folder label="Trash" />
            </folder>
         </mx:XMLList>
      </mx:XMLListCollection>
   </mx:Tree>
</mx:Application>
```

The tags that represent tree nodes in the XML data can have any name. The Tree control reads the XML and builds the display hierarchy based on the nested relationship of the nodes. For information on valid XML structure, see "Using hierarchical data providers" on page 197.

Some data providers have a single, top level called a *root* node. Other data providers are lists of nodes and do not have a root node. In some cases, you might not want to display the root node as the Tree root. To prevent the tree from displaying the root node, specify the `showRoot` property to `false`; doing this does not affect the data provider contents, only the Tree display. You can only specify a `false showRoot` property for data providers that have roots, that is, XML and Object-based data providers.

A branch node can contain multiple child nodes, and, by default, appears as a folder icon with a disclosure triangle that lets users open and close the folder. Leaf nodes appear by default as file icons and cannot contain child nodes.

When a Tree control displays a node of a non-XML data provider, by default, it displays the value of the `label` property of the node as the text label. When you use an E4X XML-based data provider, however, you must specify the label field, even if the label is identified by an attribute named "label". To specify the label field, use the `labelField` property; for example, if the label field is the label attribute, specify `labelField="@label"`.

Handling Tree control events

You typically use events to respond to user interaction with a Tree control. Since the Tree control is derived from the List control, you can use all of the events defined for the List control. The Tree control also dispatches several Event and TreeEvent class events, including `Event.change` and `TreeEvent.itemOpen`. The following example defines event handlers for the `change` and `nodeOpen` events:

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeEvents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:Script>
      <![CDATA[
         import flash.events.*;
         import mx.events.*;
         import mx.controls.*;

         private function changeEvt(event:Event):void {
            var theData:String = ""
            if (event.currentTarget.selectedItem.@data) {
              theData = " Data: " + event.currentTarget.selectedItem.@data;
            }
            forChange.text = event.currentTarget.selectedItem.@label +
               theData;
         }

      private function itemOpenEvt(event:TreeEvent):void {
         forOpen.text = event.item.@label;
      }
   ]]>
   </mx:Script>

   <mx:Tree id="XMLtree1" width="150" height="170"
          labelField="@label" itemOpen="itemOpenEvt(event);"
          change= "changeEvt(event);">
      <mx:XMLListCollection id="MailBox">
         <mx:XMLList>
            <node label="Mail" data="100">
               <node label="Inbox" data="70"/>
               <node label="Personal Folder" data="10">
                  <node label="Business" data="2"/>
```

```
                <node label="Demo" data="3"/>
                <node label="Personal" data="0" isBranch="true" />
                <node label="Saved Mail" data="5" />
            </node>
            <node label="Sent" data="15"/>
            <node label="Trash" data="5"/>
        </node>
    </mx:XMLList>
  </mx:XMLListCollection>
</mx:Tree>

<mx:Label text="Change Event:" />
<mx:TextArea id="forChange" width="150" />
<mx:Label text="Open Event:" />
<mx:TextArea id="forOpen" width="150" />
</mx:Application>
```

In this example, you define event listeners for the `change` and `itemOpen` events. The Tree control broadcasts the `change` event when the user selects a tree item, and broadcasts the `itemOpen` event when a user opens a branch node. For each event, the event handler displays the label and the data property, if any, in a TextArea control. In this example, only the Business and SavedMail nodes define a data value.

Expanding a tree node

By default, the Tree control displays the root nodes of the tree when it first opens. If you want to expand a node of the tree when the tree opens, you can use the `expandItem()` method of the Tree control. The following change to the example in "Handling Tree control events" on page 483 calls the `expandItem()` method as part of the Tree control's `creationComplete` event listener to expand the root node of the tree:

```
<mx:Script>
  <![CDATA[
.
.
.
    private function initTree():void {
      XMLTree1.expandItem(MailBox.getItemAt(0), true);
      forOpen.text=XMLTree1.openItems[0].@label;
    }
    ]]>
  </mx:Script>

<mx:Tree id="tree1" ... creationComplete="initTree();" >
  ...
</mx:Tree>
```

This example must use the Tree control's `creationComplete` event, not the `initialize` event, because the data provider is not fully initialized and available until the `creationComplete` event.

The Tree control `openItems` property is an Array containing all expanded tree nodes. The following line in the example code displays the label of the first (and only) open item in the tree:

```
forOpen.text=XMLTree1.openItems[0].@label;
```

In this example, however, you could also get the `openItems` box to indicate the initial open item by setting the `expandItem()` method to dispatch an `itemOpen` event. You can do this by specifying the fourth, optional parameter of the `expandItem()` method to `true`. The `true` fourth parameter causes the tree to dispatch an `open` event when the item opens. The following example shows the use of the fourth parameter:

```
XMLTree1.expandItem(MailBox.getItemAt(0), true, false, true);
```

Specifying Tree control icons

The Tree control provides four techniques for specifying node icons:

- The `folderOpenIcon`, `folderClosedIcon`, and `defaultLeafIcon` properties
- Data provider node icon fields
- The `setItemItcon()` method
- The `iconFunction` property

Using icon properties

You can use the `folderOpenIcon`, `folderClosedIcon`, and `defaultLeafIcon` properties to control the Tree control icons. For example, the following code specifies a default icon, and icons for the open and closed states of branch nodes:

```
<mx:Tree folderOpenIcon="@Embed(source='open.jpg')"
  folderClosedIcon="@Embed(source='closed.jpg')"
  defaultLeafIcon="@Embed(source='def.jpg')">
```

Using icon fields

You can specify an icon displayed with each Tree leaf when you populate it using XML, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeIconField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

   <mx:Script>
      <![CDATA[
         [Bindable]
         [Embed(source="assets/radioIcon.jpg")]
         public var iconSymbol1:Class;
         [Bindable]
         [Embed(source="assets/topIcon.jpg")]
         public var iconSymbol2:Class;
      ]]>
   </mx:Script>

   <mx:Tree iconField="@icon" labelField="@label" showRoot="false"
      width="160">
      <mx:XMLList>
         <node label="New">
            <node label="HTML Document" icon="iconSymbol2"/>
            <node label="Text Document" icon="iconSymbol2"/>
         </node>
         <node label="Close" icon="iconSymbol1"/>
      </mx:XMLList>
   </mx:Tree>
</mx:Application>
```

In this example, you use the `iconField` property to specify the field of each item containing the icon. You use the `Embed` metadata to import the icons, then reference them in the XML definition. You cannot specify icons for individual branch nodes; instead you must use the Tree control's `folderOpenIcon`, `folderClosedIcon` properties, each of which specifies an icon to use for all open or closed branches.

Using the setItemIcon() method

You can use the `setItemIcon()` method to specify the icon, or both the open and closed icons for a tree item. This method lets you dynamically specify and change icons for individual branches and nodes. For details on this function see setItemIcon() in *ActionScript 3.0 Language Reference*. The following example sets the open and closed node icon for the first branch node and the icon for the second branch (that does not have any leaves):

```
<?xml version="1.0"?>
<!-- dpcontrols/TreeItemIcon.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

   <mx:Script>
      <![CDATA[
         [Bindable]
         [Embed(source="assets/radioIcon.jpg")]
         public var iconSymbol1:Class;
         [Bindable]
         [Embed(source="assets/topIcon.jpg")]
         public var iconSymbol2:Class;

         private function setIcons():void {
            myTree.setItemIcon(myTree.dataProvider.getItemAt(0),
               iconSymbol1, iconSymbol2);
            myTree.setItemIcon(myTree.dataProvider.getItemAt(1),
               iconSymbol2, null);
         }
      ]]>
   </mx:Script>

   <mx:Tree id="myTree" labelField="@label" showRoot="false"
      width="160" initialize="setIcons();">
      <mx:XMLList>
         <node label="New">
            <node label="HTML Document"/>
            <node label="Text Document"/>
         </node>
         <node label="Close"/>
      </mx:XMLList>
   </mx:Tree>
</mx:Application>
```

Using an icon function

You can use the Tree control `iconFunction` property to specify a function that dynamically sets all icons for the tree. For information on using the `iconFunction` property in Flex controls, see "Specifying an icon to the List control" on page 447.

Tree user interaction

You can let users edit tree control labels. The controls also support several keyboard navigation and editing keys.

Editing a node label at run time

Set the `editable` property of the Tree control to `true` to make node labels editable at run time. To edit a node label, the user selects the label, and then enters a new label or edits the existing label text.

To support label editing, the Tree control's List superclass uses the following events. These events belong to the ListEvent class:

Event	Description
`itemEditBegin`	Dispatched when the editedItemPosition property has been set and the cell can be edited.
`itemEditEnd`	Dispatched when cell editing session ends for any reason.
`itemFocusIn`	Dispatched when tree node gets the focus: when a user selects the label or tabs to it.
`itemFocusOut`	Dispatched when a label loses focus.
`itemClick`	Dispatched when a user clicks on an item in the control.

These events are commonly used in custom item editors. For more information see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

Using the keyboard to edit labels

If you set the Tree `editable` property to `true`, you can use the following keys to edit labels:

Key	Description
Down Arrow Page Down End	Moves the caret to the end of the label.
Up Arrow Page Up Home	Moves the caret to the beginning of the label.
Right Arrow	Moves the caret forward one character.
Left Arrow	Moves the caret backwards one character.
Enter	Ends editing and moves selection to next visible node, which can then be edited. At the last node, selects the label.
Shift Enter	Ends editing and moves selection to previous visible node, which can then be edited. At the first node, selects the label.
Escape	Cancels the edit, restores the text, and changes the row state from editing to selected.
TAB	When in editing mode, accepts the current changes, selects the row below, and goes into editing mode with the label text selected. If at the last element in the tree or not in editing mode, sends focus to the next control.
Shift-TAB	When in editing mode, accepts the current changes, selects the row above, and goes into editing mode. If at the first element in the tree or not in editing mode, sends focus to the previous control.

Tree Navigation keys

When a Tree control is not editable and has focus from clicking or tabbing, you use the following keys to control it:

Key	Description
Down Arrow	Moves the selection down one. When the Tree control gets focus, use the Down arrow to move focus to the first node.
Up Arrow	Moves the selection up one item.
Right Arrow	Opens a selected branch node. If a branch is already open, moves to the first child node.
Left Arrow	Closes a selected branch node. If a leaf node or a closed branch node is currently selected, selects the parent node.
Spacebar or * (Asterisk on numeric keypad)	Opens or closes a selected branch node (toggles the state).
+ (Plus sign on numeric keypad)	Open a selected branch node.
- (Minus sign on numeric keypad)	Closes a selected branch node.
Control + Arrow keys	Move focus, but does not select a node. Use the Spacebar to select a node.
End	Moves the selection to the bottom of the list.
Home	Moves the selection to the top of the list.
Page down	Moves the selection down one page.
Page up	Moves the selection up one page.
Control	If the `allowMultipleSelection` property is `true`, allows multiple noncontiguous selections.
Shift	If the `allowMultipleSelection` property is `true`, allows multiple contiguous selections.

For information on using keys to edit tree labels, see "Using the keyboard to edit labels" on page 489.

CHAPTER 13

Introducing Containers

13

Containers provide a hierarchical structure that lets you control the layout characteristics of child components. You can use containers to control sizing and positioning of all children, or to control navigation among multiple child containers.

This topic introduces the two types of containers: layout and navigator. This topic contains an overview of container usage, including layout rules, and examples of how to use and configure containers.

Contents

About containers

A *container* defines a rectangular region of the drawing surface of Adobe Flash Player. Within a container, you define the components, both controls and containers, that you want to appear within the container. Components defined within a container are called *children* of the container. Adobe Flex provides a wide variety of containers, ranging from simple boxes through panels and forms, to elements such as accordions or tabbed navigators that provide built-in navigation among child containers.

At the root of a Flex application is a single container, called the Application container, that represents the entire Flash Player drawing surface. This Application container holds all other containers and components.

A container has predefined rules to control the layout of its children, including sizing and positioning. Flex defines layout rules to simplify the design and implementation of rich Internet applications, while also providing enough flexibility to let you create a diverse set of applications.

About container layout

Containers have predefined navigation and layout rules, so you do not have to spend time defining these. Instead, you can concentrate on the information that you deliver, and the options that you provide for your users, and not worry about implementing all the details of user action and application response. In this way, Flex provides the structure that lets you quickly and easily develop an application with a rich set of features and interactions.

Predefined layout rules also offer the advantage that your users soon grow accustomed to them. That is, by standardizing the rules of user interaction, your users do not have to think about how to navigate the application, but can instead concentrate on the content that the application offers.

Different containers support different layout rules:

- All containers, except the Canvas container, support *automatic* layout. With this type of layout you do not specify the position of the children of the container. Instead, you control the positions by selecting the container type; by setting the order of the container's children; and by specifying properties, such as gaps, and controls, such as Spacers. For example, to lay out children horizontally in a box, you use the HBox container.
- The Canvas container, and optionally the Application and Panel containers, use *absolute* layout, where you explicitly specify the children's *x* and *y* positions. Alternatively, you can use *constraint-based layout* to anchor the sides or center of the children relative to the parent.

Absolute layout provides a greater level of control over sizing and positioning than does automatic layout; for example, you can use it to overlay one control on another. But absolute layout provides this control at the cost of making you specify positions in detail.

For more information on layout, see Chapter 8, "Sizing and Positioning Components," on page 221.

About layout containers and navigator containers

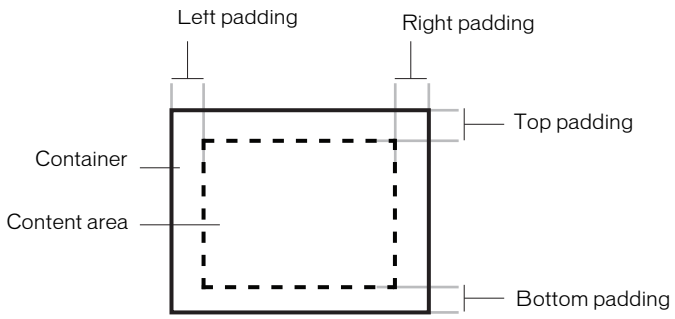
Flex defines two types of containers:

Layout containers control the sizing and positioning of the child controls and child containers defined within them. For example, a Grid layout container sizes and positions its children in a layout similar to an HTML table. Layout containers also include graphical elements that give them a particular style or reflect their function. The DividedBox container, for example, has a bar in the center that users can drag to change the relative sizes of the two box divisions. The TitleWindow control has an initial bar that can contain a title and status information. For more information on these containers, see Chapter 15, "Using Layout Containers," on page 553.

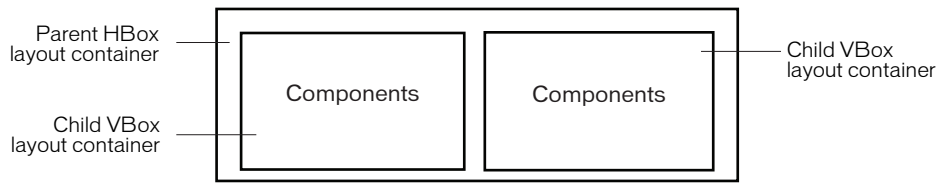
Navigator containers control user movement, or navigation, among multiple child containers. The individual child containers, not the navigator container, control the layout and positioning of their children. For example, an Accordion navigator container lets you construct a multipage form from multiple Form layout containers. For more information, see Chapter 16, "Using Navigator Containers," on page 627.

Using containers

The rectangular region of a container encloses its *content area*, the area that contains its child components. The size of the region around the content area is defined by the container padding and the width of the container border. A container has top, bottom, left, and right padding, each of which you can set to a pixel width. A container also has properties that let you specify the type and pixel width of the border. The following image shows a container and its content area, padding, and borders:



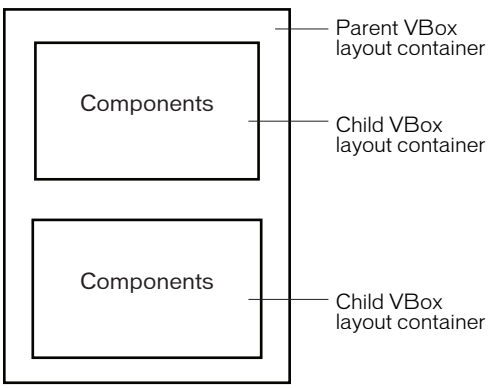
Although you can create an entire Flex application by using a single container, typical applications use multiple containers. For example, the following image shows an application that uses three layout containers:



In this example, the two VBox (vertical box) layout containers are nested within an HBox (horizontal box) layout container and are referred to as children of the HBox container.

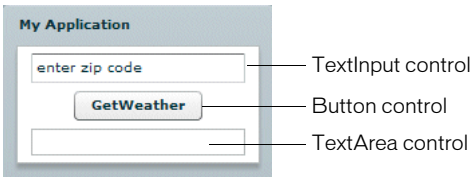
The HBox container arranges its children in a single horizontal row and oversees the sizing and positioning characteristics of the VBox containers. For example, you can control the distance, or gap, between children in a container by using the `horizontalGap` and `verticalGap` properties.

A VBox container arranges its children in a single vertical stack, or column, and oversees the layout of its own children. The following image shows the preceding example with the outermost container changed to a VBox layout container:



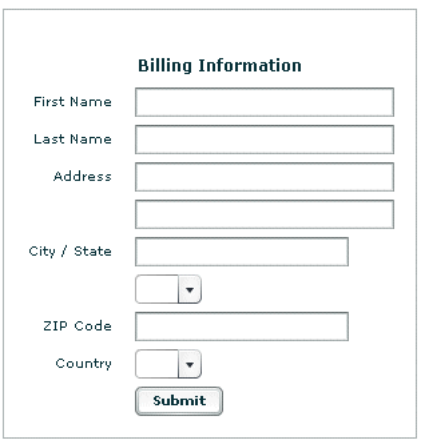
In this example, the outer container is a VBox container, so it arranges its children in a vertical column.

The primary use of a layout container is to arrange its children, where the children are either controls or other containers. The following image shows a simple VBox container that has three child components:



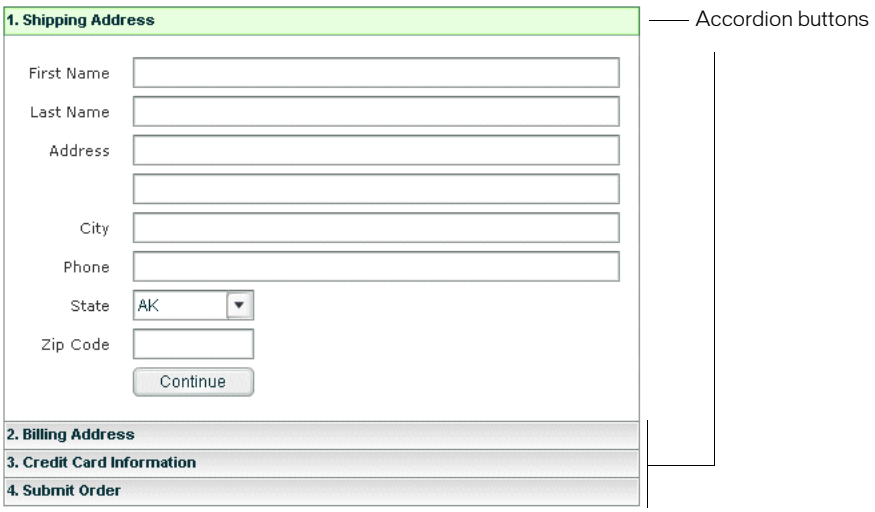
In this example, a user enters a ZIP code into the TextInput control, and then clicks the Button control to see the current temperature for the specified ZIP code in the TextArea control.

Flex supports form-based applications through its Form layout container. In a Form container, Flex can automatically align labels, uniformly size TextInput controls, and display input error notifications. The following image shows a Form container:



Form containers can take advantage of the Flex validation mechanism to detect input errors before the user submits the form. By detecting the error, and letting the user correct it before submitting the form to a server, you eliminate unnecessary server connections. The Flex validation mechanism does not preclude you from performing additional validation on the server. For more information on Form containers, see "Form, FormHeading, and FormItem layout containers" on page 570. For more information on validators, see Chapter 40, "Validating Data," on page 1281.

Navigator containers, such as the TabNavigator and Accordion containers, have built-in navigation controls that let you organize information from multiple child containers in a way that makes it easy for a user to move through it. The following image shows an Accordion container:



You use the Accordion buttons to move among the different child containers. The Accordion container defines a sequence of child panels, but displays only one panel at a time. To navigate a container, the user clicks on the navigation button that corresponds to the child panel that they want to access.

Accordion containers support the creation of multistep procedures. The preceding image shows an Accordion container that defines four panels of a complex form. To complete the form, the user enters data into all four panels. Accordion containers let users enter information in the first panel, click the Accordion button to move to the second panel, and then move back to the first if they want to edit the information. For more information, see "Accordion navigator container" on page 639.

Flex containers

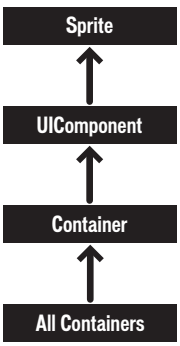
The following table describes the Flex containers:

Container	Type	Description	For more information
Accordion	Navigator	Organizes information in a series of child panels, where one panel is active at any time.	"Accordion navigator container" on page 639
Application ControlBar	Layout	Holds components that provide global navigation and application commands. Can be docked at the top of an Application container.	"ApplicationControlBar layout container" on page 564
Box (HBox and VBox)	Layout	Displays content in a uniformly spaced row or column. An HBox container horizontally aligns its children; a VBox container vertically aligns its children.	"Box, HBox, and VBox layout containers" on page 559
Canvas	Layout	Defines a container in which you must explicitly position its children.	"Canvas layout container" on page 554
ControlBar	Layout	Places controls at the lower edge of a Panel or TitleWindow container.	"ControlBar layout container" on page 562
DividedBox (HDividedBox and VDividedBox)	Layout	Lays out its children horizontally or vertically, much like a Box container, except that it inserts an adjustable divider between the children.	"DividedBox, HDividedBox, and VDividedBox layout containers" on page 567
Form	Layout	Arranges its children in a standard form format.	"Form, FormHeading, and FormItem layout containers" on page 570
Grid	Layout	Arranges children as rows and columns of cells, much like an HTML table.	"Grid layout container" on page 594
Panel	Layout	Displays a title bar, a caption, a border, and its children.	"Panel layout container" on page 601
TabNavigator	Navigator	Displays a container with tabs to let users switch between different content areas.	"TabNavigator container" on page 634

Container	Type	Description	For more information
Tile	Layout	Defines a layout that arranges its children in multiple rows or columns.	"Tile layout container" on page 606
TitleWindow	Layout	Displays a popup window that contains a title bar, a caption, border, a close button, and its children. The user can move and resize the container.	"TitleWindow layout container" on page 609
ViewStack	Navigator	Defines a stack of containers that displays a single container at a time.	"ViewStack navigator container" on page 628

Class hierarchy for containers

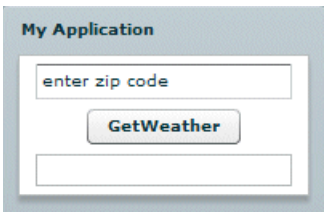
Flex containers are implemented as a hierarchy in an ActionScript class library, as the following image shows:



All containers are derived from the ActionScript classes Sprite, UIComponent, and Container, and therefore inherit the properties, methods, styles, effects, and events of their superclasses. Some containers are subclasses of other containers; for example, the ApplicationControlBar is a subclass of the ControlBar container. For a complete reference, see *Adobe Flex 2 Language Reference*.

Container example

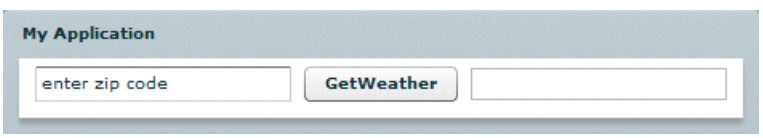
The following image shows a Flex application that uses a Panel container with three child controls, where the Panel container lays out its children vertically:



The following MXML code creates this example:

```
<?xml version="1.0"?>
<!-- containers\intro\Panel3Children.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Panel title="My Application"
        layout="vertical" horizontalAlign="center"
        paddingLeft="10" paddingRight="10"
        paddingTop="10" paddingBottom="10">
            <mx:TextInput id="myinput" text="enter zip code"/>
            <mx:Button id="mybutton" label="GetWeather"/>
            <mx:TextArea id="mytext" height="20"/>
    </mx:Panel>
</mx:Application>
```

The following image shows the preceding example implemented by using a Panel container with a horizontal layout:



The only difference between these examples is the container type and the increased width of the Application container caused by the horizontal layout, as the following code shows:

```
<?xml version="1.0"?>
<!-- containers\intro\Panel3ChildrenHoriz.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Panel title="My Application"
        layout="horizontal"
        paddingLeft="10" paddingRight="10"
        paddingTop="10" paddingBottom="10">
            <mx:TextInput id="myinput" text="enter zip code"/>
            <mx:Button id="mybutton" label="GetWeather"/>
            <mx:TextArea id="mytext" height="20"/>
    </mx:Panel>
</mx:Application>
```

To actually retrieve weather information, you must set up a web service, pass it the entered ZIP code from a `click` event, and use the returned information to populate the TextArea control.

Using container events

All containers and components support events, as described in the following sections.

Event overview

The following events are dispatched only by containers:

- `childAdd` Dispatched after a child is added to the container.
- `childRemove` Dispatched before a child is removed from the container.
- `childIndexChange` Dispatched after a child's index in the container has changed.
- `scroll` Dispatched when the user manually scrolls the container.

The first three events are dispatched for each of the container's children, and the last is dispatched when the container scrolls. For detailed information on these events, see Container in *Adobe Flex 2 Language Reference*.

The following events are dispatched only by Application containers:

- `applicationComplete` Dispatched after the application has been initialized, processed by the LayoutManager, and attached to the display list. This is the last event dispatched during an application's startup sequence. It is later than the application's `creationComplete` event, which gets dispatched before the preloader has been removed and the application has been attached to the display list.
- `error` Dispatched when an uncaught error occurs anywhere in the application.

The following events are dispatched by all components after they are added to or removed from a container:

- `add` Dispatched by a component after the component has been added to its container and the parent and the child are in a consistent state. This event is dispatched after the container has dispatched the `childAdd` event and all changes that need to be made as result of the addition have happened.
- `remove` Dispatched by a component after the component has been removed from its parent container. This event is dispatched after the container has dispatched the `childRemove` event and all changes that need to be made as result of the removal have happened.

Several events are dispatched for all components, but need special consideration for containers, particularly navigator containers such as TabNavigator, where some children might not be created when the container is created. These events include the following:

- `preinitialize` Dispatched when the component has been attached to its parent container, but before the component has been initialized, or any of its children have been created. In most cases, this event is dispatched too early for an application to use to configure a component.
- `initialize` Dispatched when a component has finished its construction and its initialization properties have been set. At this point, all of the component's immediate children have been created (they have at least dispatched their `preinitialize` event), but they have not been laid out. Exactly when initialize events are dispatched depends on the container's creation policy, as described later in this section.
- `creationComplete` Dispatched when the component, and all of its child components, and all of their children, and so on have been created, laid out, and are visible.

For information on the `initialize` and `creationComplete` events, see "About the initialize and creationComplete events" on page 502. For information on the remaining events, see `Container` in *Adobe Flex 2 Language Reference*.

About the creation policy

Containers have a `creationPolicy` property that specifies when its children are created. By default, containers have an creation policy of `ContainerCreationPolicy.AUTO`, which means that the container delays creating descendants until they are needed, a process which is known as deferred instantiation. (A container's default creation policy is derived from its parent's instantiation policy, and the Application container has a default policy of `ContainerCreationPolicy.AUTO`.

An auto creation policy produces the best startup time because fewer components are created initially. For navigator containers such as the ViewStack, TabNavigator, and Accordion, an `ContainerCreationPolicy.AUTO` creation policy means that the container creates its direct children immediately, but waits to create the descendants of each child until the child needs to be displayed. As a result, only the initially required child or children of a container get processed past the preinitialization stage.

A creation policy of `ContainerCreationPolicy.ALL` requires all of a container's children to be fully created and initialized before the container is initialized. For detailed information on creation policies, see Chapter 6, "Improving Startup Performance," in *Building and Deploying Flex 2 Applications*.

About the initialize and creationComplete events

Flex dispatches the `initialize` event for a container after it attaches all the container's direct child controls and the container's initially required children have dispatched a `preinitialize` event.

When a container or control dispatches the `initialize` event, its initial properties have been set, but its width and height have not yet been calculated, and its position has not been calculated. The `initialize` event is useful for configuring a container's children. For example, you can use the a container's `initialize` event to programmatically add children or set a container scroll bar's styles. You can use a container or component's initialize `initialize` event to initialize the data provider for a control.

Flex dispatches the `creationComplete` event for a container when those children that are initially required are fully processed and drawn on the screen, including all required children of the children and so on. Create a listener for the `creationComplete` event, for example, if you must have the children's dimensions and positions in your event handler. Do not use the `creationComplete` event for actions that set layout properties, as doing so results in excess processing time.

To better understand the order in which Flex dispatches events, consider the following application outline.

```
Application
  OuterVBox
    InnerVBox1
      InnerVBoxLabel1
    InnerVBox2
      InnerVBoxLabel2
```

The `preinitialize`, `initialize`, and `creationComplete` events for the containers and controls are dispatched in the following order. The indentation corresponds to the indentation in the previous outline:

```
OuterVBox  preinitialize
  InnerVBox1  preinitialize
    InnerVBox1Label  preinitialize
    InnerVBox1Label  initialize
  InnerVBox1  initialize
  InnerVBox2  preinitialize
    InnerVBox2Label  preinitialize
    InnerVBox2Label  initialize
  InnerVBox2  initialize
OuterVBox  initialize
    InnerBox1Label  creationComplete
    InnerVBox2Label  creationComplete
  InnerVBox1  creationComplete
  InnerVBox2  creationComplete
OuterVBox  creationComplete
```

Notice that for the terminal controls, such as the Label controls, the controls are preinitialized and then immediately initialized. For containers, preinitialization starts with the outermost container and works inward on the first branch, and then initialization works outward on the same branch. This process continues until all initialization is completed. Then, the `creationComplete` event is dispatched first by the leaf components, and then by their parents, and so on until the application dispatches the `creationComplete` event.

If you change the OuterVBox container to a ViewStack with a `creationPolicy` property set to `auto`, the events would look as follows:

```
OuterViewStack  preinitialize
  InnerVBox1  preinitialize
  InnerVBox2  preinitialize
OuterViewStack  initialize
    InnerBox1Label  preinitialize
    InnerBox1Label  initialize
  InnerVBox1  initialize
    InnerBox1Label  creationComplete
  InnerVBox1  creationComplete
OuterViewStack  creationComplete
```

In this case, the second VBox is preinitialized only, because it does not have to be displayed. Notice that when a navigator container dispatches the `initialize` event, its children exist and have dispatched the `preinitialize` event, but its children have not dispatched the `initialize` event because they have not yet created their own children. For more information on the `creationPolicy` property, see Chapter 6, "Improving Startup Performance," in *Building and Deploying Flex 2 Applications*.

The `initialize` event is useful with a container that is an immediate child of a navigator container with an `ContainerCreationPolicy.AUTO` creation policy. For example, by default, when a ViewStack is initialized, the first visible child container dispatches an `initialize` event. Then, as the user moves to each additional child of the container, the event gets dispatched for that child container.

The following example defines an event listener for the `initialize` event, which is dispatched when the user first navigates to panel 2 of an Accordion container:

```
<?xml version="1.0"?>
<!-- containers\intro\AccordionInitEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;

            public function pane2_initialize():void {
                Alert.show("Pane 2 has been created");
            }
        ]]>
    </mx:Script>

    <mx:Accordion width="200" height="100">
        <mx:VBox id="pane1" label="Pane 1">
            <mx:Label text="This is pane 1"/>
        </mx:VBox>
        <mx:VBox id="pane2"
            label="Pane 2"
            initialize="pane2_initialize();">
            <mx:Label text="This is pane 2"/>
        </mx:VBox>
    </mx:Accordion>
</mx:Application>
```

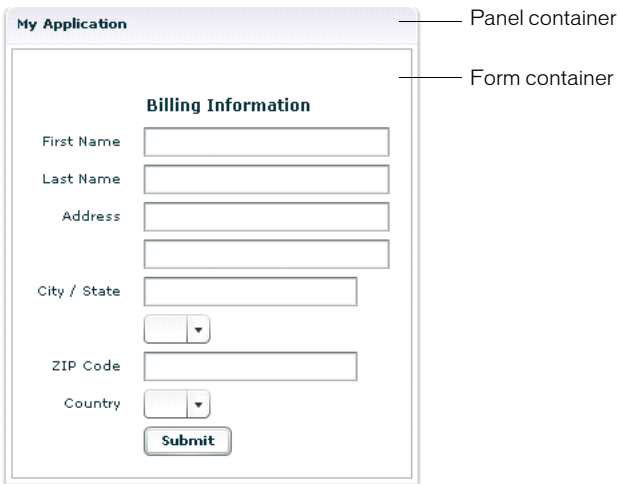
Disabling containers

All containers support the `enabled` property. By default, this property is set to `true` to enable user interaction with the container and with the container's children. If you set `enabled` to `false`, Flex dims the color of the container and of all of its children, and blocks user input to the container and to all of its children.

Using the Panel container

One container that you often use in a Flex application is the Panel container. The Panel container consists of a title bar, a caption, a status message, a border, and a content area for its children. Typically, you use a Panel container to wrap self-contained application modules. For example, you can define several Panel containers in your application where one Panel container holds a form, a second holds a shopping cart, and a third holds a shopping catalog. The Flex RichTextEditor control is a Panel control that contains a TextArea control and a ControlBar control that has editing controls.

The following image shows a Panel container with a Form container as its child:



For more information on the Panel container, see "Panel layout container" on page 601.

You can also define a ControlBar control as part of a Panel container. A ControlBar control defines an area at the lower edge of the Panel container, below any children in the Panel container.

You can use the ControlBar container to hold components that might be shared by the other children in the Panel container, or for controls that operate on the content of the Panel container. For example, you can use the ControlBar container to display the subtotal of a shopping cart, where the shopping cart is defined in the Panel container. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart. For more information on the ControlBar container, see "ControlBar layout container" on page 562.

Defining a default button

You use the `defaultButton` property of a container to define a default Button control within a container. Pressing the Enter key while focus is on any control activates the Button control as if it was explicitly selected.

For example, a login form displays TextInput controls for a user name and password and a submit Button control. Typically, the user types a user name, tabs to the password field, types the password, and presses the Enter key to submit the login information without explicitly selecting the Button control. To define this type of interaction, set the `defaultButton` property of the Form control to the `id` of the submit Button control, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerDefaultB.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function submitLogin():void {
                text1.text="You just tried to log in";
            }
        ]]>
    </mx:Script>

    <mx:Panel title="Default Button Example">

        <mx:Form defaultButton="{mySubmitBtn}">
            <mx:FormItem label="Username">
                <mx:TextInput id="username"
                    width="100"/>
            </mx:FormItem>
            <mx:FormItem label="Password">
                <mx:TextInput id="password"
                    width="100"
                    displayAsPassword="true"/>
            </mx:FormItem>
            <mx:FormItem>
                <mx:Button id="mySubmitBtn"
                    label="Login"
                    click="submitLogin();"/>
            </mx:FormItem>
        </mx:Form>
        <mx:Text id="text1" width="150"/>
    </mx:Panel>
</mx:Application>
```

NOTE

The Enter key has a special purpose in the ComboBox control. When the drop-down list of a ComboBox control is open, pressing Enter selects the currently highlighted item in the ComboBox control; it does not activate the default button. Also, when the cursor is in a TextArea control, pressing Enter adds a newline; it does not activate the default button.

Using scroll bars

Flex containers support scroll bars, which let you display an object that is larger than the available screen space or display more objects than fit in the current size of the container, as the following image shows:

Image at full size

Image in an HBox container

In this example, you use an HBox container to let users scroll an image, rather than rendering the complete image at its full size:

```
<?xml version="1.0"?>
<!-- containers\intro\HBoxScroll.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="75" height="75">
        <mx:Image source="assets/logo.jpg"/>
    </mx:HBox>
</mx:Application>
```

In this example, you explicitly set the size of the HBox container to 75 by 75 pixels, a size smaller than the imported image. If you omit the sizing restrictions on the HBox container, it attempts to use its default size, which is a size large enough to hold the image.

By default, Flex draws scroll bars only when the contents of a container are larger than that container. To force the container to draw scroll bars, you can set the `horizontalScrollPolicy` and `verticalScrollPolicy` properties to `on`.

The following example creates an HBox container with scroll bars even though the image inside is large enough to display fully without them:

```
<?xml version="1.0"?>
<!-- containers\intro\HBoxScrollOn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox horizontalScrollPolicy="on" verticalScrollPolicy="on">
        <mx:Image source="assets/logo.jpg"/>
    </mx:HBox>
</mx:Application>
```

Using container scroll properties

The following container properties and styles control scroll bar appearance and behavior:

- The `horizontalScrollPolicy` and `verticalScrollPolicy` properties control the display of scroll bars. By default, both properties are set to `auto`, which configures Flex to include scroll bars only when necessary. You can set these properties to `on` to configure Flex to always include scroll bars, or set the properties to `off` to configure Flex to never include scroll bars. In ActionScript, you can use constants in the ScrollPolicy class, such as `ScrollPolicy.ON`, to represent these values.
- The `horizontalLineScrollSize` and `verticalLineScrollSize` properties determine how many pixels to scroll when the user selects the scroll bar arrows. The default value is 5 pixels.
- The `horizontalPageScrollSize` and `verticalPageScrollSize` properties determine how many pixels to scroll when the user selects the scroll bar track. The default value is 20 pixels.

NOTE

If the `clipContent` property is `false`, a container lets its child extend past its boundaries. Therefore, no scroll bars are necessary, and Flex never displays them, even if you set `horizontalScrollPolicy` and `verticalScrollPolicy` to `on`.

Scroll bar layout considerations

Your configuration of scroll bars can affect the layout of your application. For example, if you set the `horizontalScrollPolicy` and `verticalScrollPolicy` properties to `on`, the container always includes scroll bars, even if they are not necessary. Each scroll bar is 16 pixels wide. Therefore, turning them on when they are not needed is similar to increasing the size of the right and bottom padding of the container by 16 pixels.

If you keep the default values of `auto` for the `horizontalScrollPolicy` and `verticalScrollPolicy` properties, Flex lays out the application just as if the properties are set to `off`. That is, the scroll bars are not counted as part of the layout.

If you do not keep this behavior in mind, your application might have an inappropriate appearance. For example, if you have an HBox container that is 30 pixels high and 100 pixels wide and has two buttons that are each 22 pixels high and 40 pixels wide, the children are contained fully inside the HBox container, and no scroll bars appear. However, if you add a third button, the children exceed the width of the HBox container, and Flex adds a horizontal scroll bar at the bottom of the container. The scroll bar is 16 pixels high, which reduces the height of the content area of the container from 30 pixels to 14 pixels. This means that the Button controls, which are 22 pixels high, are too tall for the HBox, and Flex, by default, adds a vertical scroll bar.

Controlling scroll delay and interval

Scroll bars have two styles that affect how they scroll:

- The `repeatDelay` style specifies the number of milliseconds to wait after the user selects a scroll button before repeating scrolling.
- The `repeatInterval` style specifies the number of milliseconds to wait between each repeated scroll while the user keeps the scroll arrows selected.

These settings are styles of the scroll bar subcontrol, not of the container, and, therefore, require a different treatment than properties such as `horizontalScrollPolicy`. The following example sets the scroll policy consistently for all scroll bars in the application:

```
<?xml version="1.0"?>
<!-- containers\intro\HBoxScrollDelay.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        HScrollBar, VScrollBar {
            repeatDelay: 2000;
            repeatInterval:1000;
        }
    </mx:Style>

    <mx:HBox id="hb1" width="75" height="75">
        <mx:Image source="adobe_logo.gif"/>
    </mx:HBox>
</mx:Application>
```

This example results in the same scrollable logo as shown in "Using scroll bars" on page 507, but the scroll bars behave differently. When the user clicks and holds the mouse button down over any of the scroll bar arrows or the scroll bar track, the image initially scrolls once, waits two seconds, and then scrolls at a rate of one line or page a second.

To set a style on a single scroll bar, use a line such as the following in the event listener for the `initialize` event for the application or the control with the scroll bar:

```
ScrollBar(hb1.horizontalScrollBar).setStyle("repeatDelay", 2000);
```

In this case, `hb1` is an HBox control. All containers have `horizontalScrollBar` and `verticalScrollBar` properties that represent the container's ScrollBar subcontrols, if they exist. You must cast these properties to the ScrollBar class, because their type is the IScrollBar interface, not the ScrollBar class.

Using Flex coordinates

Adobe Flash and Flex support three coordinate systems for different purposes:

- global
- local
- content

The following sections describe the coordinate systems, when they are used, and when and how to use Flex properties and methods to convert between coordinate systems.

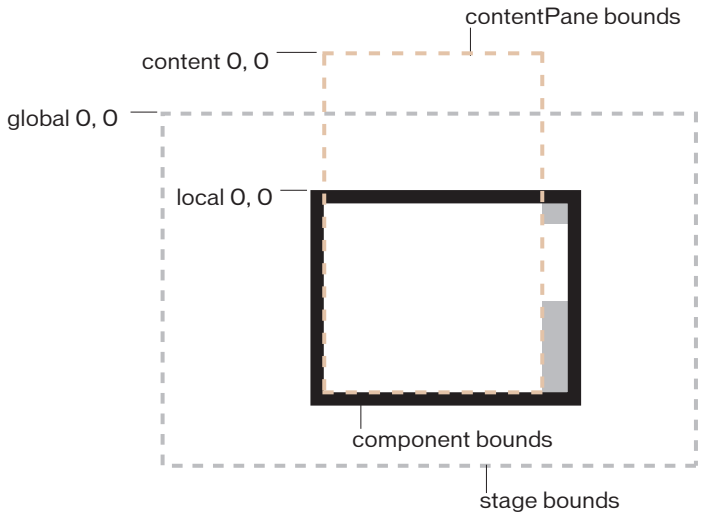
About the coordinate systems

The following table describes the coordinate systems:

Coordinate system	Description
global	Coordinates are relative to the upper-left corner of the Stage in Adobe Flash Player, that is, the outermost edge of the application. The global coordinate system provides a universal set of coordinates that are independent of the component context. Uses for this coordinate system include determining distances between objects and as an intermediate point in converting between coordinates relative to a subcontrol into coordinates relative to a parent control. The MouseEvent class includes `stageX` and `stageY` properties that are in the global coordinate system.

Coordinate system	Description
local	Coordinates are relative to the upper-left corner of the component. Flex uses the local coordinate system for mouse pointer locations; all components have `mouseX` and `mouseY` properties that use the local coordinate system. The MouseEvent class includes localX and localY properties that are in the local coordinate system. Also, the Drag Manager uses local coordinates in drag-and-drop operations. The `doDrag()` method's `xOffset` and `yOffset` properties, for example, are offsets relative to the local coordinates.
content	Coordinates are relative to the upper-left corner of the component's content. Unlike the local and global coordinates, the content coordinates include all of the component's content area, including any regions that are currently clipped and must be accessed by scrolling the component. Thus, if you scrolled down a Canvas container by 100 pixels, the upper-left corner of the visible content is at position 0, 100 in the content coordinates. You use the content coordinate system to set and get the positions of children of a container that uses absolute positioning. (For more information on absolute positioning, see "About component positioning" on page 224.) The UIComponent `contentMouseX` and `contentMouseY` properties report the mouse pointer location in the content coordinate system.

The following image shows these coordinate systems and how they relate to each other.



Using coordinate properties and methods

In some cases, you have to convert positions between coordinate systems. Examples where you convert between coordinates include the following:

- The MouseEvent class has properties that provide the mouse position in the global coordinate system and the local coordinates of the event target. You use the content coordinates to specify the locations in a Canvas container, or Application or Panel container that uses absolute positioning. To determine the location of the mouse event within the Canvas container contents, not just the visible region, you must determine the position in the content coordinate system.
- Custom drag-and-drop handlers might have to convert between the local coordinate system and the content coordinate system when determining an object-specific drag action; for example, if you have a control with scroll bars and you want to know the drag (mouse) location over the component contents. The example in "Example: Using the mouse position in a Canvas container" on page 513 shows this use.
- Custom layout containers, where you include both visual elements, such as scroll bars or dividers, and content elements. For example, if you have a custom container that draws lines between its children, you have to know where each child is in the container's content coordinates to draw the lines.

Often, you use mouse coordinates in event handlers; when you do, you should keep the following considerations in mind:

- When you handle mouse events, it is best to use the coordinates from the MouseEvent object whenever possible, because they represent the mouse coordinates at the time the event was generated. Although you can use the container's `contentMouseX` and `contentMouseY` properties to get the mouse pointer locations in the content coordinate system, you should, instead, get the local coordinate values from the event object and convert them to the content coordinate system.
- When you use local coordinates that are reported in an event object, such as the MouseEvent `localX` and `localY` properties, you must remember that the event properties report the local coordinates of the mouse relative to the event target. The target component can be a subcomponent of the component in which you determine the position, such as a UITextField inside a Button component, not the component itself. In such cases, you must convert the local coordinates into the global coordinate system first, and then convert the global coordinates into the content coordinates container.

Coordinate conversion properties and methods

All Flex components provide two read-only properties and six functions that enable you to use and convert between coordinate systems. The following table describes these properties and functions:

Property or function	Description
`contentMouseX`	Returns the *x* position of the mouse, in the content coordinates of the component.
`contentMouseY`	Returns the *y* position of the mouse, in the content coordinates of the component.
`contentToGlobal (point:Point):Point`	Converts a Point object with *x* and *y* coordinates from the content coordinate system to the global coordinate system.
`contentToLocal (point:Point):Point`	Converts a Point object from the content coordinate system to the local coordinate system of the component.
`globalToContent (point:Point):Point`	Converts a Point object from the global coordinate system to the content coordinate system of the component.
`globalToLocal (point:Point):Point`	Converts a Point object from the global coordinate system to the local coordinate system of the component.
`localToContent (point:Point):Point`	Converts a Point object from the local coordinate system to the content coordinate system of the component.
`localToGlobal (point:Point):Point`	Converts a Point object from the local coordinate system to the global coordinate system.

Example: Using the mouse position in a Canvas container

The following example shows the use of the `localToGlobal()` and `globalToContent()` methods to determine the location of a mouse pointer within a Canvas container that contains multiple child Canvas containers.

This example is somewhat artificial, in that production code would use the MouseEvent class `stageX` and `stageY` properties, which represent the mouse position in the global coordinate system. The example uses the `localX` and `localY` properties, instead, to show how you can convert between local and content coordinates, including how first converting to using the global coordinates ensures the correct coordinate frame of reference.

```
<?xml version="1.0"?>
<!-- containers\intro\MousePosition.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="white">

    <mx:Script>
        <![CDATA[
          import mx.controls.Alert;

          // Handle the mouseDown event generated
          // by clicking in the application.
          private function handleMouseDown(event:MouseEvent):void {

            // Convert the mouse position to global coordinates.
            // The localX and localY properties of the mouse event contain
            // the coordinates at which the event occurred relative to the
            // event target, typically one of the
            // colored internal Canvas controls.
            // A production version of this example could use the stageX
            // and stageY properties, which use the global coordinates,
            // and avoid this step.
            // This example uses the localX and localY properties only to
            // illustrate conversion between different frames of reference.
            var pt:Point = new Point(event.localX, event.localY);
            pt = event.target.localToGlobal(pt);

            // Convert the global coordinates to the content coordinates
            // inside the outer c1 Canvas control.
            pt = c1.globalToContent(pt);

            // Figure out which quadrant was clicked.
            var whichColor:String = "border area";

            if (pt.x < 150) {
                if (pt.y < 150)
                    whichColor = "red";
                else
                    whichColor = "blue";
            }
            else {
                if (pt.y < 150)
                    whichColor = "green";
                else
```

```
                    whichColor = "magenta";
                }

                Alert.show("You clicked on the " + whichColor);
              }
            ]]>
        </mx:Script>

        <!-- Canvas container with four child Canvas containers -->
        <mx:Canvas id="c1"
            borderStyle="none"
            width="300" height="300"
            mouseDown="handleMouseDown(event);">

            <mx:Canvas
                width="150" height="150"
                x="0" y="0"
                backgroundColor="red">
                <mx:Button label="I'm in Red"/>
            </mx:Canvas>
            <mx:Canvas
                width="150" height="150"
                x="150" y="0"
                backgroundColor="green">
                <mx:Button label="I'm in Green"/>
            </mx:Canvas>
            <mx:Canvas
                width="150" height="150"
                x="0" y="150"
                backgroundColor="blue">
                <mx:Button label="I'm in Blue"/>
            </mx:Canvas>
            <mx:Canvas
                width="150" height="150"
                x="150" y="150"
                backgroundColor="magenta">
                <mx:Button label="I'm in Magenta"/>
            </mx:Canvas>
        </mx:Canvas>
    </mx:Application>
```

Creating and managing component instances at run time

You typically use MXML to lay out the user interface of your application, and use ActionScript for event handling and run-time control of the application. You can also use ActionScript to create component instances at run time. For example, you could use MXML to define an empty Accordion container and use ActionScript to add panels to the container in response to user actions.

About the display list and container children

Flash Player maintains a tree of visible (or potentially visible) objects that make up your application. The root of the tree is the Application object, and child containers and components are branches and leaf nodes of the tree. That tree is known as the *display list*. When you add child components to a container or remove child components from a container, you are adding and removing them from the display list. You can also change their relative positions by changing their positions in the display list.

Although the display list is a tree rooted at the top of the application, when you manipulate a container's children in ActionScript by using the container's methods and properties, you only access the container's direct children, and you treat them as items in a list with an index that starts at 0 for the container's first child in the display list.

The Container class includes the `numChildren` property, which contains a count of the number of direct child components that the container has in the display list. The following HBox container, for example, includes two child components, so the value of its `numChildren` property is 2:

```
<mx:HBox id="myContainer">
  <mx:Button click="clickHandler();"/>
  <mx:TextInput/>
</mx:HBox>
```

You can access and modify a container's child components at run time by using the `addChild()`, `addChildAt()`, `getChildren()`, `getChildAt()`, `getChildByName()`, `removeAllChildren()`, `removeChild()`, and `removeChildAt()` methods of the Container class. For example, you can iterate over all the child component of a container, as the following example shows:

```
private function clickHandler():void {
  var numChildren:Number = myContainer.numChildren;
  for (var i:int = 0; i < numChildren; i++) {
    trace(myContainer.getChildAt(i));
  }
}
```

The Container class also defines the `rawChildren` property that contains the full display list of all of the children of a container. This list includes all the container's children, plus the DisplayObjects that implement the container's *chrome* (display elements), such as its border and the background image. For more information see "Accessing display-only children" on page 519.

Obtaining the number of child components in a container or application

To get the number of direct child components in a container, get the value of the container's `numChildren` property. The following application gets the number of children in the application and a VBox container. The VBox control has five label controls, and therefore has five children. The Application container has the VBox and Button controls as its children, and therefore has two children.

```
<?xml version="1.0"?>
<!-- containers\intro\VBoxNumChildren.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
      <![CDATA[

            // Import the Alert class.
            import mx.controls.Alert;

            public function calculateChildren():void {
                var myText:String = new String();
                myText="The VBox container has " +
                    myVBox.numChildren + " children";
                myText+="\nThe Application has " +
                    numChildren + " children";
                Alert.show(myText);
            }
        ]]>
    </mx:Script>

    <mx:VBox id="myVBox" borderStyle="solid">
        <mx:Label text="This is label 1"/>
        <mx:Label text="This is label 2"/>
        <mx:Label text="This is label 3"/>
        <mx:Label text="This is label 4"/>
        <mx:Label text="This is label 5"/>
    </mx:VBox>

    <mx:Button label="Show Children" click="calculateChildren();"/>
</mx:Application>
```

In the main MXML application file, the file that contains the `<mx:Application>` tag, the current scope is always the Application object. Therefore, the reference to the `numChildren` property without an object prefix refers to the `numChildren` property of the Application object. For more information on accessing the root application, see "About scope" on page 66.

Accessing display-only children

The `numChildren` property and `getChildAt()` method let you count and access only child components. Also, the container may contain style elements and skins, such as the border and background. The container's `rawChildren` property lets you access all children of a container, including the component "content children" and the skin and style "display children." The object returned by the `rawChildren` property implements the IChildList interface. You then use methods and properties of this interface, such as `getChildAt()`, to access and manipulate all the container's children.

Creating and removing components at run time

To create a component instance at run time, you define it, set any properties, and then add it as a child of a parent container by calling the `addChild()` method on the parent container. This method has the following signature:

```
addChild(child:DisplayObject):DisplayObject
```

The `child` argument specifies the component to add to the container.

NOTE

Although the `child` argument of the method is specified as type DisplayObject, the argument must implement the IUIComponent interface to be added as a child of a container. All Flex components implement this interface.

For example, the following application creates an HBox container with a Button control called myButton:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerAddChild.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Button;

            public function addButton():void {
                var myButton:Button = new Button();
                myButton.label = "New Button";
                myHBox.addChild(myButton);
            }
        ]]>
    </mx:Script>

    <mx:HBox id="myHBox" initialize="addButton();"/>
</mx:Application>
```

This example creates the control when the application is loaded rather than in response to any user action. However, you could add a new button when the user presses an existing button, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerAddChild2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="left">

    <mx:Script>
        <![CDATA[
            import mx.controls.Button;

            public function addButton():void {
                var myButton:Button = new Button();
                myButton.label = "New Button";
                myHBox.addChild(myButton);
            }
        ]]>
    </mx:Script>

    <mx:HBox id="myHBox">
        <mx:Button label="Add Button" click="addButton();"/>
    </mx:HBox>
</mx:Application>
```

The following image shows the resulting application after the user presses the original (leftmost) button three times:

You use the `removeChild()` method to remove a control from a container. Flex sets the `parent` property of the removed child. If the child is no longer referenced anywhere else in your application after the call to the `removeChild()` method, it gets destroyed by a garbage collection process. The following example removes a button from the application when the user presses it:

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerRemoveChild.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function removeButton():void {
                myHBox.removeChild(myButton);
            }
        ]]>
    </mx:Script>

    <mx:HBox id="myHBox">
        <mx:Button id="myButton"
            label="Remove Me"
            click="removeButton();"/>
    </mx:HBox>
</mx:Application>
```

For additional methods that you can use with container children, see the Container class in *Adobe Flex 2 Language Reference*.

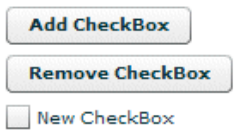
Example: Creating and removing a child of an VBox container

The following example uses MXML to defines a VBox container that contains two Button controls. You use one Button control to add a CheckBox control to the VBox container, and one Button control to delete it.

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerComponentsExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
      <![CDATA[

        // Import the CheckBox class.
        import mx.controls.CheckBox;

        // Define a variable to hold the new CheckBox control.
        private var myCheckBox:CheckBox;

        // Define a variable to track if the CheckBox control
        // is in the display list.
        private var checkBoxDisplayed:Boolean = false;

        public function addCB():void {
            // Make sure the check box isn't being displayed.
            if(checkBoxDisplayed==false){
                // Create the check box if it does not exist.
                if (!myCheckBox) {
                    myCheckBox = new CheckBox();
                }

                // Add the check box.
                myCheckBox.label = "New CheckBox";
                myVBox.addChild(myCheckBox);
                checkBoxDisplayed=true;
            }
        }

        public function delCB():void {
            // Make sure a CheckBox control exists.
            if(checkBoxDisplayed){
                myVBox.removeChild(myCheckBox);
                checkBoxDisplayed=false;
            }
        }
      ]]>
    </mx:Script>

    <mx:VBox id="myVBox">
        <mx:Button label="Add CheckBox"
```

```
            click="addCB();"/>
        <mx:Button label="Remove CheckBox"
            click="delCB();"/>
    </mx:VBox>
</mx:Application>
```

The following image shows the resulting application after the user presses the Add CheckBox button:



Example: Creating and removing children of an Accordion container

The example in this section adds and removes panels to an Accordion container. The Accordion container initially contains one panel. Each time you select the Add HBox button, it adds a new HBox container to the Accordion container.

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerComponentsExample2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Import HBox class.
            import mx.containers.HBox;

            //Array of created containers
            private var hBoxes:Array = [];

            public function addHB():void {
                // Create new HBox container.
                var newHB:HBox = new HBox();
                newHB.label="my label: " + String(hBoxes.length);

                // Add it to the Accordion container, and to the
                // Array of HBox containers.
                hBoxes.push(myAcc.addChild(newHB));
            }

            public function delHB():void {
                // If there is at least one HBox container in the Array,
                // remove it.
                if (hBoxes.length>= 1) {
                    myAcc.removeChild(hBoxes.pop());
                }
            }
      ]]>
    </mx:Script>

    <mx:VBox>
        <mx:Accordion id="myAcc" height="150" width="150">
            <mx:HBox label="Initial HBox"/>
        </mx:Accordion>

        <mx:Button label="Add HBox" click="addHB();"/>
        <mx:Button label="Remove HBox" click="delHB();"/>
    </mx:VBox>
</mx:Application>
```

Controlling child order

You can control the order of children by adding them in a specific order. You can also control them as follows:

- By using the `addChildAt()` method to specify where among the component's children to add a child
- By using the `setChildIndex()` method to specify the location of a specific child among a component's children in the display list

NOTE

As with the `addChild()` method, although the `child` argument of the `addChildAt()` method is specified as type DisplayObject, the argument must implement the IUIComponent interface to be added as a child of a container. All Flex components implement this interface.

The following example modifies the example in "Example: Creating and removing a child of an VBox container" on page 522. It uses the `addChildAt()` method to add the CheckBox control as the first child (index 0) of the VBox. It also has a Reorder children button that uses the `setChildIndex()` method to move a CheckBox control down the display list until it is the last the child in the VBox container. Boldface text indicates the lines that are added to or changed from the previous example.

```
<?xml version="1.0"?>
<!-- containers\intro\ContainerComponentsReorder.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
      <![CDATA[

        // Import the CheckBox and Alert classes.
        import mx.controls.CheckBox;
        import mx.controls.Alert;

        // Define a variable to hold the new CheckBox control.
        private var myCheckBox:CheckBox;

        // Define a variable to track if the CheckBox control
        // is in the display list.
        private var checkBoxDisplayed:Boolean = false;

        public function addCB():void {
            // Make sure the check box isn't being displayed.
            if(checkBoxDisplayed==false){
                // Create the check box if it does not exist.
                if (!myCheckBox) {
                    myCheckBox = new CheckBox();
                }
                // Add the check box as the first child of the VBox.
                myCheckBox.label = "New CheckBox";
                myVBox.addChildAt(myCheckBox, 0);
                checkBoxDisplayed=true;
            }
        }

        public function delCB():void {
            // Make sure a CheckBox control exists.
            if(checkBoxDisplayed){
                myVBox.removeChild(myCheckBox);
                checkBoxDisplayed=false;
            }
        }

        public function reorder():void {
        // Make sure a CheckBox control exists.
```

```
            if(checkBoxDisplayed==true){
                // Don't try to move the check box past the end
                // of the children. Because indexes are 0 based,
                // the last child index is one less
                // than the number of children.
               if (myVBox.getChildIndex(myCheckBox) < myVBox.numChildren-1)
                {
                    // Increment the checkBoxIndex variable and use it to
                  // set the index of the check box among the VBox children.
                    myVBox.setChildIndex(myCheckBox,
                        myVBox.getChildIndex(myCheckBox) + 1);
                }
            }
            else {
                Alert.show("Add the check box before you can move it");
            }
        }
      ]]>
    </mx:Script>

    <mx:VBox id="myVBox">
        <mx:Button label="Add CheckBox" click="addCB();"/>
        <mx:Button label="Remove CheckBox" click="delCB();"/>
        <mx:Button label="Reorder children" click="reorder();"/>
    </mx:VBox>
</mx:Application>
```

CHAPTER 14

Using the Application Container

14

Adobe Flex defines a default Application container that lets you start adding content to your application without having to explicitly define another container. This topic describes how to use an Application container.

Flex defines any MXML file that contains an `<mx:Application>` tag as an Application object. For more information, see "About the Application object" on page 537.

The Application container supports an application preloader that uses a progress bar to show the download progress of an application SWF file. You can override the default progress bar to define your own custom progress bar. For more information, see "Showing the download progress of an application" on page 542.

Contents

Using the Application container

Flex defines an Application container that serves as the default container for any content that you add to your application. Flex creates this container from the `<mx:Application>` tag, which must be the first tag in an MXML application file. The Application object is the default scope for any ActionScript code in the file, and the `<mx:Application>` tag defines the initial size of the application.

Although you may find it convenient to use the Application container as the only container in your application, usually you explicitly define at least one more container before you add any controls to your application. Often, you use a Panel container as the first container after the `<mx:Application>` tag.

The Application container has the following default layout characteristics:

Property	Default value
Default size	The size of the browser window
Child alignment	Vertical column arrangement of children
Child horizontal alignment	Centered
Default padding	24 pixels for the `top`, `bottom`, `left`, and `right` properties

Sizing an Application container and its children

An Application container arranges its children in a single vertical column. You can set the height and width of the Application container by using explicit pixel values or by using percentage values, where the percentage values are relative to the size of the browser window. By default, the Application container has a height and width of 100%, which means that it fills the entire browser window.

The following example sets the size of the Application container to one-half of the width and height of the browser window:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  height="50%" width="50%">
  ...
</mx:Application>
```

The advantage of using percentages to specify the size is that Flex can resize your application as the user resizes the browser window. Flex maintains the Application container size as a percentage of the browser window as the user resizes it.

If you set the `width` and `height` properties of the child components MXML tags to percentage values, your components can also resize as your application resizes, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppSizePercent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%">

    <mx:Panel title="Main Application" width="100%" height="100%">
        <mx:HDividedBox width="100%" height="100%">
            <mx:TextArea width="50%" height="100%"/>
            <mx:VDividedBox width="50%" height="100%">
                <mx:DataGrid width="100%" height="25%"/>
                <mx:TextArea width="100%" height="75%"/>
            </mx:VDividedBox>
        </mx:HDividedBox>
    </mx:Panel>
</mx:Application>
```

The following example uses explicit pixel values to size the Application container:

```
<?xml version="1.0"?>
<!-- containers\application\AppSizePixel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="100" width="150">

    <mx:Panel title="Main Application">
        <mx:TextInput id="mytext" text="Hello"/>
        <mx:Button id="mybutton" label="Get Weather"/>
    </mx:Panel>
</mx:Application>
```

If the children of the Application container are sized or positioned such that some or all of the component is outside of the visible area of the Application container, Flex adds scroll bars to the container, as the preceding example shows.

If you want to set a child container to fill the entire Application container, the easiest method is to set the child's MXML tag `width` and `height` properties to 100% (or, in ActionScript, set the `percentWidth` and `percentHeight` properties to 100), and set the Application container padding to 0. If you base the child container's `width` and `height` properties on those of the Application container, you must subtract the Application container's padding, or your child container will be larger than the available space, and the application will have scroll bars.

In the following example, the VBox container expands to fill all the available space, except for the area defined by the Application container padding:

```
<?xml version="1.0"?>
<!-- containers\application\AppVBoxSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100" height="100">

    <mx:VBox width="100%" height="100%" backgroundColor="#A9C0E7">
        <!-- ... -->
    </mx:VBox>
</mx:Application>
```

In the following example, the VBox container is larger than the available space within the Application container, which results in scroll bars:

```
<?xml version="1.0"?>
<!-- containers\application\AppVBoxSizeScroll.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100" height="100">

    <mx:VBox width="200" height="200" backgroundColor="#A9C0E7">
        <!-- ... -->
    </mx:VBox>
</mx:Application>
```

In the following example, the Application container has no padding, which lets its child VBox container fill the entire window:

```
<?xml version="1.0"?>
<!-- containers\application\AppNoPadding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100" height="100"
    paddingTop="0" paddingBottom="0"
    paddingLeft="0" paddingRight="0">

    <mx:VBox width="100" height="100" backgroundColor="#A9C0E7">
        <!-- ... -->
    </mx:VBox>
</mx:Application>
```

Overriding the default Application container styles

By default, the Application container has the following default style properties that define the following visual aspects of a Flex application and differ from the default container values:

Property	Default value
`backgroundColor`	The color of the Stage area of Adobe Flash Player, which is visible during application loading and initialization. This color is also visible if the application background is transparent. The default value is `0x869CA7`.
`backgroundGradientAlphas`	`[1.0, 1.0]`, a fully opaque background.
`backgroundGradientColors`	`[0x9CB0BA, 0x68808C]`, a grey background that is slightly darker at the bottom.
`backgroundImage`	A gradient controlled by the `backgroundGradientAlphas` and `backgroundGradientColors` styles. The default value is `mx.skins.halo.ApplicationBackground`.
`backgroundSize`	100%. When you set this property at 100%, the background image takes up the entire Application container.
`horizontalAlign`	Centered.
`paddingBottom`	24 pixels.
`paddingLeft`	24 pixels.
`paddingRight`	24 pixels.
`paddingTop`	24 pixels.

You can override these default values in your application to define your own default style properties.

Changing the Application background

The Application container `backgroundGradientAlphas`, `backgroundGradientColors`, and `backgroundImage` styles control the container background. By default, these properties define an opaque grey gradient background.

You specify an image for the application background by using the `backgroundImage` property. If you set both the `backgroundImage` property and the `backgroundGradientColors` property, Flex ignores `backgroundGradientColors`.

You can specify a gradient background for the application in two ways:

- Set the `backgroundGradientColors` property to two values, as in the following example:

  ```
  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundGradientColors="[0x0000FF, 0xCCCCCC]">
  ```

  Flex calculates the gradient pattern between the two specified values.

- Set the `backgroundColor` property to the desired value, as in the following example:

  ```
  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="red">
  ```

  Flex calculates the gradient pattern between a color slightly darker than red, and a color slightly lighter than red.

To set a solid background to the application, specify the same two values to the `backgroundGradientColors` property, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  backgroundGradientColors="[#FFFFFF, #FFFFFF]">
```

This example defines a solid white background.

The `backgroundColor` property specifies the background color of the Stage area in Flash Player, which is visible during application loading and initialization, and a background gradient while the application is running. By default, the `backgroundColor` property is set to `0x869CA7`, which specifies a dark blue-grey color.

If you use the `backgroundGradientColors` property to set the application background, you should also set the `backgroundColor` property to compliment the `backgroundGradientColors` property, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  backgroundGradientColors="[0x0000FF, 0xCCCCCC]"
  backgroundColor="0x0000FF">
```

In this example, you use the `backgroundGradientColors` property to set a gradient pattern from a dark blue to grey, and the `backgroundColor` property to set the Stage area in Flash Player to dark blue, which will be visible during application loading and initialization.

Using the plain style

The Flex default style sheet defines a plain style name that sets all padding to 0 pixels, removes the default background image, sets the background color to white, and left-aligns the children. The following example shows how you can set this style:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  styleName="plain">
```

You can override individual values in the plain setting, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  styleName="plain" horizontalAlign="center"/>
```

Overriding styles with the Style tag

You can also use the `<mx:Style>` tag in your application to specify alternative style values, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppStyling.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Style definition for the entire application. -->
    <mx:Style>
        Application {
            paddingLeft: 10px;
            paddingRight: 10px;
            paddingTop: 10px;
            paddingBottom: 10px;
            horizontalAlign: "left";
            backgroundImage: "";
            backgroundColor: #AAAACC;
        }
    </mx:Style>

    <mx:Panel title="Main Application">
        <mx:TextInput id="mytext" text="Hello"/>
        <mx:Button id="mybutton" label="Get Weather"/>
    </mx:Panel>
</mx:Application>
```

This example removes the background image, sets all padding to 10 pixels, left-aligns children, and sets the background color to a light blue.

For more information on using styles, see Chapter 18, “Using Styles and Themes,” on page 697.

Viewing the application source code

You can use the `viewSourceURL` property of the Application container to specify a URL to the application’s source code. If you set this property, Flex adds a View Source menu item to the application’s context menu, which you open by right-clicking anywhere in your application. Select the View Source menu item to open the URL specified by the `viewSourceURL` property in a new browser window.

You must set the `viewSourceURL` property by using MXML, not ActionScript, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppSourceURL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    viewSourceURL="http://localhost:8100/flex/assets/AppSourceURL.txt">

    <mx:Button/>
</mx:Application>
```

You typically deploy your source code not as an MXML file but as a text or HTML file. In this example, the source code is in the file AppSourceURL.txt. If you use an HTML file to represent your source code, you can add formatting and coloring to make it easier to read.

Specifying options of the Application container

You can specify several options of the `<mx:Application>` tag to control your application. The following table describes these options:

Option	Type	Description
`frameRate`	Number	Specifies the frame rate of the application, in frames per second. The default value is 24.
`pageTitle`	String	Specifies a String that appears in the title bar of the browser. This property provides the same functionality as the HTML `<title>` tag.
`preloader`	Path	Specifies the path of a SWC component class or ActionScript component class that defines a custom progress bar. A SWC component must be in the same directory as the MXML file or in the WEB-INF/flex/user_classes directory of your Flex web application. For more information, see "Showing the download progress of an application" on page 542.
`scriptRecursionLimit`	Number	Specifies the maximum depth of the Flash Player call stack before Flash Player stops. This is essentially the stack overflow limit. The default value is 1000.

Option	Type	Description
`scriptTimeLimit`	Number	Specifies the maximum duration, in seconds, that an ActionScript event listener can execute before Flash Player assumes that it has stopped processing and aborts it. The default value is 60 seconds, which is also the maximum allowable value that you can set.
`usePreloader`	Boolean	Specifies whether to disable the application preloader (`false`) or not (`true`). The default value is `true`. To use the default preloader, your application must be at least 160 pixels wide. For more information, see "Showing the download progress of an application" on page 542.

About the Application object

Flex compiles your application into a SWF file that contains a single Application object, defined by the `<mx:Application>` tag. In most cases, your Flex application has one Application object. Some applications use the SWFLoader control to add more applications.

An Application object has the following characteristics:

- Application objects are MXML files with an `<mx:Application>` tag.
- Most Flex applications have a single Application object.
- The Application file is the first file loaded.
- An Application object is also a Document object, but a Document object is not always an Application object. For more information on the Document object, see "About the Document object" on page 537.
- You can refer to the Application object as `mx.core.Application.application` from anywhere in the Flex application.
- If you load multiple nested applications by using the SWFLoader control, you can access the scope of each higher application in the nesting hierarchy by using `parentApplication`, `parentApplication.parentApplication`, and so on.

About the Document object

Flex creates a Document object for every MXML file used in a Flex application. For example, you can have a document, which is also an Application object, and from there, use other MXML files that define custom controls.

A Document object has the following characteristics:

- All MXML files that a Flex application uses are Document objects, including the Application object's file.
- Custom ActionScript component files are Document objects.
- The Flex compiler cannot compile a SWF file from a file that does not contain an `<mx:Application>` tag.
- Documents usually consist of MXML custom controls that you use in your Flex application.
- You can access the scope of a document's parent document by using `parentDocument`, `parentDocument.parentDocument`, and so on.
- Flex provides a `UIComponent.isDocument` property so that you can detect if any given object is a Document object.

Accessing Document and Application object scopes

In your application's main MXML file, the file that contains the `<mx:Application>` tag, you can access the methods and properties of the Application object using the `this` keyword. However, in custom ActionScript and MXML components, event listeners, or external ActionScript class files, Flex executes in the context of those components and classes, and the `this` keyword refers to the current Document object and not the Application object. You cannot refer to a control or method in the application from one of these child documents without specifying the location of the parent document.

Flex provides the following properties that you can use to access parent documents:

mx.core.Application.application The top-level Application object, regardless of where in the document tree your object executes.

mx.core.UIComponent.parentDocument The parent document of the current document. You can use `parentDocument.parentDocument` to walk up the tree of multiple documents.

mx.core.UIComponent.parentApplication The Application object in which the current object exists. Flex applications can load applications into applications, therefore, you can access the immediate parent application by using this property. You can use `parentApplication.parentApplication` to walk up the tree of multiple applications.

The following sections describe how to use these properties.

Using the mx.core.Application.application property

To access properties and methods of the top-level Application object from anywhere in your application, you can use the `application` property of the Application class. For example, you define an application that contains the doSomething() method, as the following code shows:

```
<?xml version="1.0"?>
<!-- containers\application\AppDoSomething.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComps="myComponents.*">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;

            // Open an Alert control.
            public function doSomething():void {
                Alert.show("doSomething() called.");
            }
        ]]>
    </mx:Script>

    <!-- Include the ButtonMXML.mxml component. -->
    <MyComps:ButtonMXML/>
</mx:Application>
```

You can then use the `Application.application` property in the ButtonMXML.mxml component to reference the doSomething() method, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\myComponents\ButtonMXML.mxml -->
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // To refer to the members of the Application class,
            // you must import mx.core.Application.
            import mx.core.Application;
        ]]>
    </mx:Script>

    <mx:Button  label="MXML Button"
        click="Application.application.doSomething();"/>
</mx:HBox>
```

The `application` property is especially useful in applications that have one or more custom MXML or ActionScript components that each use a shared set of data. At the application level, you often store shared information and provide utility functions that any of the components can access.

For example, suppose that you store the user's name at the application level and implement a utility function, `getSalutation()`, which returns the string "Hi, *userName*". The following example MyApplication.mxml file shows the application source that defines the `getSalutation()` method:

```
<?xml version="1.0"?>
<!-- containers\application\AppSalutation.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComps="myComponents.*">

    <mx:Script>
        <![CDATA[
            public var userName:String="SMG";
            public function getSalutation():String {
                return "Hi, " + userName;
            }
        ]]>
    </mx:Script>

    <!-- Include the ButtonGetSalutation.mxml component. -->
    <MyComps:ButtonGetSalutation/>

</mx:Application>
```

To access the `userName` and call the `getSalutation()` method in your MXML components, you can use the `application` property, as the following example from the MyComponent.mxml component shows:

```
<?xml version="1.0"?>
<!-- containers\application\myComponents\ButtonGetSalutation.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%" >

    <mx:Script>
        <![CDATA[
            // To refer to the members of the Application class,
            // you must import mx.core.Application.
            import mx.core.Application;
        ]]>
    </mx:Script>

    <mx:Label id="myL"/>
    <mx:Button
        click="myL.text=Application.application.getSalutation();"/>
</mx:VBox>
```

In this example, clicking the Button control executes the `getSalutation()` function to populate the Label control.

Using the parentDocument property

To access the parent document of an object, you can use the `parentDocument` property. The parent document is the object that contains the current object. All classes that inherit from the UIComponent class have a `parentDocument` property.

In the following example, the application references the custom AccChildObject.mxml component:

```
<?xml version="1.0"?>
<!-- containers\application\AppParentDocument.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComps="myComponents.*">

    <!-- Include the AccChildObject.mxml component. -->
    <MyComps:AccChildObject/>

</mx:Application>
```

In this example, the application is the parent document of the AccChildObject.mxml component. The following code from the AccChildObject.mxml component uses the `parentDocument` property to define an Accordion container that is slightly smaller than the Application container:

```
<?xml version="1.0"?>
<!-- containers\application\myComponents\AccChildObject.mxml -->
<mx:Accordion xmlns:mx="http://www.adobe.com/2006/mxml"
    width="{parentDocument.width*.80}"
    height="{parentDocument.height*.50}">

    <mx:HBox/>

</mx:Accordion>
```

You use the `parentDocument` property in MXML scripts to go up a level in the chain of parent documents. You can use the `parentDocument` to walk this chain by using multiple `parentDocument` properties, as the following example shows:

```
parentDocument.parentDocument.doSomething();
```

The `parentDocument` property of the Application object is a reference to the application.

The `parentDocument` is typed as Object so that you can access properties and methods on ancestor Document objects without casting.

Every UIComponent class has an `isDocument` property that is set to `true` if that UIComponent class is a Document object, and `false` if it is not.

If a UIComponent class is a Document object, it has a `documentDescriptor` property. This is a reference to the descriptor at the top of the generated descriptor tree in the generated Document class.

For example, suppose that AddressForm.mxml component creates a subclass of the Form container to define an address form, and the MyApp.mxml component creates two instances of it: `<AddressForm id="shipping">` and `<AddressForm id="billing">`.

In this example, the shipping object is a Document object. Its `documentDescriptor` property corresponds to the `<mx:Form>` tag at the top of the AddressForm.mxml file (the definition of the component), while its descriptor corresponds to the `<AddressForm id="shipping">` tag in MyApp.mxml file (an instance of the component).

Walking the document chain by using the `parentDocument` property is similar to walking the application chain by using the `parentApplication` property.

Using the parentApplication property

Applications can load other applications; therefore, you can have a hierarchy of applications, similar to the hierarchy of documents within each application. Every UIComponent class has a `parentApplication` read-only property that references the Application object in which the object exists. The `parentApplication` property of an Application object is never itself; it is either the Application object into which it was loaded, or it is `null` (for the Application object).

Walking the application chain by using the `parentApplication` property is similar to walking the document chain by using the `parentDocument` property.

Showing the download progress of an application

The Application class supports an application preloader that uses a download progress bar to show the download progress of an application SWF file. By default, the application preloader is enabled. The preloader keeps track of how many bytes have been downloaded and continually updates the progress bar.

The download progress bar displays information about two different phases of the application: the download phase and the initialization phase. The `Application.creationComplete` event dismisses the preloader.

The following example shows the download progress bar during the initialization phase:

The download progress bar is not displayed if the SWF file is on your local host or if it is already cached. If the SWF file is not on your local host and is not cached, the progress bar is displayed if less than half of the application is downloaded after 700 milliseconds of downloading.

Disabling the download progress bar

To disable the download progress bar, you set the `usePreloader` property of the Application container to `false`, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  usePreloader="false">
```

Creating a custom progress bar

By default, the application preloader uses the DownloadProgressBar class in the mx.preloaders package to display the download progress bar. To create a custom download progress bar, you can either create a subclass of the DownloadProgressBar class, or create a subclass of the flash.display.Sprite class that implements the mx.preloaders.IPreloaderDsiplay interface.

You can implement a download progress bar component as a SWC component or an ActionScript component. A custom download progress bar component that extends the Sprite class should not use any of the standard Flex components because it would load too slowly to be effective. Do not implement a download progress bar as an MXML component because it also would load too slowly.

To use a custom download progress bar class, you set the `preloader` property of the Application container to the path of a SWC component class or ActionScript component class. A SWC component must be in the same directory as the MXML file or in a directory on the classpath of your Flex application. An ActionScript component can be in one of those directories or in a subdirectory of one of those directories. When a class is in a subdirectory, you specify the subdirectory location as the package name in the `preloader` value; otherwise, you specify the class name.

The code in the following example specifies a custom download progress bar called CustomBar that is located in the mycomponents/mybars directory below the application's root directory:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  preloader="mycomponents.mybars.CustomBar">
```

Download progress bar events

The operation of the download progress bar is defined by a set of events. These events are dispatched by the Preloader class. A custom download progress bar must handle these events.

The following table describes the download progress bar events:

Event	Description
`ProgressEvent.PROGRESS`	Dispatched when the application SWF file is being downloaded. The first `PROGRESS` event signifies the beginning of the download process.
`Event.COMPLETE`	Dispatched when the SWF file has finished downloading. Either zero or one `COMPLETE` event is dispatched.
`FlexEvent.INIT_COMPLETE`	Dispatched when the Flex application finishes initialization. This event is always dispatched once, and is the last event that the Preloader dispatches. The download progress bar must dispatch a `COMPLETE` event after it has received an `INIT_COMPLETE` event. The `COMPLETE` event informs the Preloader that the download progress bar has completed all operations and can be dismissed. The download progress bar can perform additional tasks, such as playing an animation, after receiving an `INIT_COMPLETE` event, and before dispatching the `COMPLETE` event. Dispatching the `COMPLETE` event should be the last action of the download progress bar.
`FlexEvent.INIT_PROGRESS`	Dispatched when the Flex application completes an initialization phase, as defined by calls to the `measure()`, `commitProperties()`, or `updateDisplayList()` methods. This event describes the progress of the application in the initialization phase.
`RslEvent.RSL_ERROR`	Dispatched when a Runtime Shared Library (RSL) fails to load.
`RslEvent.RSL_LOADED`	Dispatched when an RSL finishes loading. The total bytes and total loaded bytes are included in the event object. This event is dispatched for every RSL that is successfully loaded.
`RSLEvent.RSL_PROGRESS`	Dispatched when an RSL is being downloaded. The first progress event signifies the beginning of the RSL download. The event object for this event is of type RSLEvent.

The DownloadProgressBar class defines an event listener for all of these events. Within your override of the DownloadProgressBar class, you can optionally override the default behavior of the event listener. If you create a custom download progress bar as a subclass of the Sprite class, you must define an event listener for each of these events.

Creating a simple subclass of the DownloadProgressBar class

The easiest way to create your own download progress bar is to create a subclass of the mx.preloaders.DownloadProgressBar class, and then modify it for your application requirements.

Your example might define custom strings for the download progress bar, or set the minimum time that it appears, as the following example shows:

```
package myComponents
{
    import mx.preloaders.*;
    import flash.events.ProgressEvent;

    public class DownloadProgressBarSubClassMin extends DownloadProgressBar
    {
        public function DownloadProgressBarSubClassMin()
        {
            super();
            // Set the download label.
            downloadingLabel="Downloading app..."
            // Set the initialization label.
            initializingLabel="Initializing app..."
            // Set the minimum display time to 2 seconds.
            MINIMUM_DISPLAY_TIME=2000;
        }

        // Override to return true so progress bar appears
        // during initialization.
        override protected function showDisplayForInit(elapsedTime:int,
            count:int):Boolean {
                return true;
        }

      // Override to return true so progress bar appears during download.
        override protected function showDisplayForDownloading(
            elapsedTime:int, event:ProgressEvent):Boolean {
                return true;
        }
    }

}
```

You can use your custom class in a Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\MainDPBMin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    preloader="myComponents.DownloadProgressBarSubClassMin">

    <mx:Button/>
    <mx:TextInput text="sub class min" />
</mx:Application>
```

Creating a subclass of the DownloadProgressBar class

In the following example, you create a subclass of the DownloadProgressBar class to display text messages that describe the status of the downloading and initialization of the application. This example defines event listeners for the events dispatched by the download progress bar to write the messages to flash.text.TextField objects.

```
package myComponents
{

    import flash.display.*;
    import flash.text.*;
    import flash.utils.*;
    import flash.events.*;
    import mx.preloaders.*;
    import mx.events.*;

    public class MyDownloadProgressBar extends DownloadProgressBar
    {
        // Define a TextField control for text messages
        // describing the download progress of the application.
        private var progressText:TextField;

        // Define a TextField control for the final text message.
        // after the application initializes.
        private var msgText:TextField;

        public function MyDownloadProgressBar()
        {
            super();

            // Configure the TextField for progress messages.
            progressText = new TextField();
            progressText.x = 10;
            progressText.y = 90;
            progressText.width = 400;
            progressText.height = 400;

            addChild(progressText);

            // Configure the TextField for the final message.
            msgText = new TextField();
            msgText.x = 10;
            msgText.y = 10;
            msgText.width = 400;
            msgText.height = 75;

            addChild(msgText);
```

```
        }

        // Define the event listeners for the preloader events.
        override public function set preloader(preloader:Sprite):void {
            // Listen for the relevant events
            preloader.addEventListener(
                ProgressEvent.PROGRESS, myHandleProgress);
            preloader.addEventListener(
                Event.COMPLETE, myHandleComplete);

            preloader.addEventListener(
                FlexEvent.INIT_PROGRESS, myHandleInitProgress);
            preloader.addEventListener(
                FlexEvent.INIT_COMPLETE, myHandleInitEnd);
        }

        // Event listeners for the ProgressEvent.PROGRESS event.
        private function myHandleProgress(event:ProgressEvent):void {
            progressText.appendText("\n" + "Progress l: " +
                event.bytesLoaded + " t: " + event.bytesTotal);
        }

        // Event listeners for the Event.COMPLETE event.
        private function myHandleComplete(event:Event):void {
            progressText.appendText("\n" + "Completed");
        }

        // Event listeners for the FlexEvent.INIT_PROGRESS event.
        private function myHandleInitProgress(event:Event):void {
            progressText.appendText("\n" + "App Init Start");
        }

        // Event listeners for the FlexEvent.INIT_COMPLETE event.
        private function myHandleInitEnd(event:Event):void {
            msgText.appendText("\n" + "App Init End");

            var timer:Timer = new Timer(2000,1);
            timer.addEventListener(TimerEvent.TIMER, dispatchComplete);
            timer.start();
        }

        // Event listener for the Timer to pause long enough to
        // read the text in the download progress bar.
        private function dispatchComplete(event:TimerEvent):void {
            dispatchEvent(new Event(Event.COMPLETE));
        }
    }
}
```

You can use your custom class in a Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\application\MainDPB.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    preloader="myComponents.MyDownloadProgressBar">

    <mx:Button/>
    <mx:TextInput/>
</mx:Application>
```

Creating a subclass of Sprite

You can define a custom download progress bar as a subclass of the Sprite class. By implementing your download progress bar as a subclass of Sprite, you can create a completely custom look and feel to it, rather than overriding the behavior built into the DownloadProgressBar class.

One common use for this type of download progress bar is to have it display a SWF file during application initialization. For example, you could display a SWF file that shows a running clock, or other type of image.

The example in this section display a SWF file as the download progress bar. This class must implement the IPreloaderDisplay interface.

```
package myComponents
{
    import flash.display.*;
    import flash.utils.*;
    import flash.events.*;
    import flash.net.*;
    import mx.preloaders.*;
    import mx.events.*;

    public class MyDownloadProgressBarSWF extends Sprite
        implements IPreloaderDisplay
    {
        // Define a Loader control to load the SWF file.
        private var dpbImageControl:flash.display.Loader;

        public function MyDownloadProgressBarSWF() {
            super();
        }

        // Specify the event listeners.
        public function set preloader(preloader:Sprite):void {
            // Listen for the relevant events
            preloader.addEventListener(
                ProgressEvent.PROGRESS, handleProgress);
            preloader.addEventListener(
                Event.COMPLETE, handleComplete);

            preloader.addEventListener(
                FlexEvent.INIT_PROGRESS, handleInitProgress);
            preloader.addEventListener(
                FlexEvent.INIT_COMPLETE, handleInitComplete);
        }

        // Initialize the Loader control in the override
        // of IPreloaderDisplay.initialize().
        public function initialize():void {
            dpbImageControl = new flash.display.Loader();
            dpbImageControl.contentLoaderInfo.addEventListener(
                Event.COMPLETE, loader_completeHandler);
          dpbImageControl.load(new URLRequest("assets/dpbSWF.swf"));
        }

        // After the SWF file loads, set the size of the Loader control.
        private function loader_completeHandler(event:Event):void
        {
            addChild(dpbImageControl);
            dpbImageControl.width = 50;
```

```
    dpbImageControl.height= 50;
    dpbImageControl.x = 100;
    dpbImageControl.y = 100;
}

// Define empty event listeners.
private function handleProgress(event:ProgressEvent):void {
}

private function handleComplete(event:Event):void {
}

private function handleInitProgress(event:Event):void {
}

private function handleInitComplete(event:Event):void {
    var timer:Timer = new Timer(2000,1);
    timer.addEventListener(TimerEvent.TIMER, dispatchComplete);
    timer.start();
}

private function dispatchComplete(event:TimerEvent):void {
    dispatchEvent(new Event(Event.COMPLETE));
}

// Implement IPreloaderDisplay interface

public function get backgroundColor():uint {
    return 0;
}

public function set backgroundColor(value:uint):void {
}

public function get backgroundAlpha():Number {
    return 0;
}

public function set backgroundAlpha(value:Number):void {
}

public function get backgroundImage():Object {
    return undefined;
}

public function set backgroundImage(value:Object):void {
}

public function get backgroundSize():String {
    return "";
```

```
        }

        public function set backgroundSize(value:String):void {
        }

        public function get stageWidth():Number {
            return 200;
        }

        public function set stageWidth(value:Number):void {
        }

        public function get stageHeight():Number {
            return 200;
        }

        public function set stageHeight(value:Number):void {
        }
    }
}
```

CHAPTER 15

Using Layout Containers

15

In Adobe Flex, layout containers provide a hierarchical structure to arrange and configure the components, such as Button and ComboBox controls, of a Flex application.

This topic describes the layout containers and their usage, and includes descriptions and examples of all Flex layout containers. For detailed information on how Flex lays out containers and their children, see Chapter 8, "Sizing and Positioning Components," on page 221.

Contents

About layout containers

A layout container defines a rectangular region of the Adobe Flash Player drawing surface and controls the sizing and positioning of the child controls and child containers defined within it. For example, a Form layout container sizes and positions its children in a layout similar to an HTML form.

To use a layout container, you create the container, and then add the components that define your application.

Flex provides the following layout containers:

- Canvas layout container
- Box, HBox, and VBox layout containers
- ControlBar layout container
- ApplicationControlBar layout container
- DividedBox, HDividedBox, and VDividedBox layout containers
- Form, FormHeading, and FormItem layout containers
- Grid layout container
- Panel layout container
- Tile layout container
- TitleWindow layout container

The following sections describe how to use each of the Flex layout containers.

Canvas layout container

A Canvas layout container defines a rectangular region in which you place child containers and controls. Unlike all other components, you cannot let Flex lay child controls out automatically. You must use *absolute* or *constraint-based* layout to position child components. With absolute layout you specify the *x* and *y* positions of the children; with constraint-based layout you specify side or center anchors. For detailed information on using these layout techniques, see Chapter 8, "Sizing and Positioning Components," on page 221.

The Canvas container has the following default sizing characteristics:

Property	Default value
Default size	Large enough to hold all its children at the default sizes of the children
Default padding	0 pixels for the `top`, `bottom`, `left`, and `right` values

For complete reference information, see Canvas in *Adobe Flex 2 Language Reference*.

Creating and using a Canvas control

You define a canvas control in MXML using the `<mx:Canvas>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

Creating a Canvas Control by using absolute positioning

You can use the `x` and `y` properties of each child to specify the child's location in the Canvas container. These properties specify the *x* and *y* coordinates of a child relative to the upper-left corner of the Canvas container, where the upper-left corner is at coordinates (0,0). Values for the *x* and *y* coordinates can be positive or negative integers. You can use negative values to place a child outside the visible area of the container, and then use ActionScript to move the child to the visible area, possibly as a response to an event.

The following example shows a Canvas container with three LinkButton controls and three Image controls:

The following MXML code creates this Canvas container:

```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Canvas id="myCanvas"
      height="200" width="200"
      borderStyle="solid"
      backgroundColor="white">

      <mx:LinkButton label="Search"
        x="10" y="30"
        click="navigateToURL(new URLRequest('http://mycomp.com/search'))"/>
      <mx:Image
        height="50" width="50"
        x="100" y="10"
        source="@Embed(source='assets/search.jpg')"
        click="navigateToURL(new URLRequest('http://mycomp.com/search'))"/>

      <mx:LinkButton label="Help"
        x="10" y="100"
        click="navigateToURL(new URLRequest('http://mycomp.com/help'))"/>
      <mx:Image
        height="50" width="50"
        x="100" y="75"
        source="@Embed(source='assets/help.jpg')"
        click="navigateToURL(new URLRequest('http://mycomp.com/help'))"/>

      <mx:LinkButton label="Complaints"
        x="10" y="170"
        click="navigateToURL(
            new URLRequest('http://mycomp.com/complain'))"/>
      <mx:Image
        height="50" width="50"
        x="100" y="140"
        source="@Embed(source='assets/complaint.jpg')"
        click="navigateToURL(
            new URLRequest('http://mycomp.com/complaint'))"/>
    </mx:Canvas>
</mx:Application>
```

Creating a Canvas container by using constraint-based layout

You can also use *constraint-based layout* to anchor any combination of the top, left, right, and bottom sides of a child a specific distance from the Canvas edges, or to anchor the horizontal or vertical center of the child a specific (positive or negative) pixel distance from the Canvas center. To specify a constraint-based layout you use the `top`, `bottom`, `left`, `right`, `horizontalCenter`, and `verticalCenter` styles. When you anchor the top and bottom, or the left and right sides of the child container to the Canvas sides, if the Canvas control resizes, the children also resize. The following example uses constraint-based layout to position an HBox horizontally, and uses absolute values to specify the vertical width and position:

```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasConstraint.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Canvas
        width="150" height="150"
        backgroundColor="#FFFFFF">

        <mx:HBox id="hBox2"
            left="30"
            right="30"
            y="50"
            height="50"
            backgroundColor="#A9C0E7">
        </mx:HBox>
    </mx:Canvas>
</mx:Application>
```

The example produces the following image:

Preventing overlapping children

When you use a Canvas container, some of your components may overlap, because the Canvas container ignores its children's sizes when it positions them. Similarly, children components may overlap any borders or padding, because the Canvas container does not adjust the coordinate system to account for them.

In the following example, the size and position of each component is carefully calculated to ensure that none of the components overlap:

```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasOverlap.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100" height="100"
    backgroundGradientColors="[0xFFFFFF, 0xFFFFFF]">

    <mx:Canvas id="chboard" backgroundColor="#FFFFFF">
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="0" y="0"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="20" y="0"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="40" y="0"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="10" y="10"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="30" y="10"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="0" y="20"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="20" y="20"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="40" y="20"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="10" y="30"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="30" y="30"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="0" y="40"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="20" y="40"/>
        <mx:Image source="assets\BlackBox.jpg"
            width="10" height="10" x="40" y="40"/>
    </mx:Canvas>
</mx:Application>
```

This example produces the following image:

If you set the `width` and `height` properties of one of the images to 20 pixels but don't change the positions accordingly, that image overlaps other images in the checkerboard. For example, if you replace the seventh `<mx:Image>` tag in the preceding example with the following line, the resulting image looks like the following image:

```
<mx:Image source="BlackBox.jpg" width="10" height="10" x="20" y="20"/>
```

Repositioning children at run time

You can build logic into your application to reposition a child of a Canvas container at run time. For example, in response to a button click, the following code repositions an input text box that has the `id` value `text1` to the position x=110, y=110:

```
<?xml version="1.0"?>
<!-- containers\layouts\CanvasOverlap.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Canvas
        width="300" height="300"
        backgroundColor="#FFFFFF">

        <mx:TextInput id="text1"
            text="Move me"
            x="50" y="50"/>
        <mx:Button id="button1"
            label="Move text1"
            x="50" y="200"
            click="text1.x=110; text1.y=110;"/>
    </mx:Canvas>
</mx:Application>
```

Box, HBox, and VBox layout containers

The Box layout container lays out its children in a single vertical column or a single horizontal row. You use the `direction` property of a Box container to determine either vertical (default) or horizontal layout. The HBox and VBox containers are Box containers with `horizontal` and `vertical direction` property values.

NOTE

To lay out children in multiple rows or columns, use a Tile or Grid container. For more information, see "Tile layout container" on page 606 and "Grid layout container" on page 594.

The following example shows one Box container with a horizontal layout and one with a vertical layout:

Box container with horizontal layout

Box container with vertical layout (default)

A Box container has the following default sizing characteristics:

Property	Default value
Default size	**Vertical Box** The height is large enough to hold all its children at the default or explicit height of the children, plus any vertical gap between the children, plus the top and bottom padding of the container. The width is the default or explicit width of the widest child, plus the left and right padding of the container. **Horizontal Box** The height is the default or explicit height of the tallest child, plus the top and bottom padding for the container. The width is large enough to hold all of its children at the default width of the children, plus any horizontal gap between the children, plus the left and right padding of the container.
Default padding	0 pixels for the `top`, `bottom`, `left`, and `right` values.

For complete reference information, see Box, HBox, and VBox in *Adobe Flex 2 Language Reference*.

Creating a Box, HBox, or VBox container

You use the `<mx:Box>`, `<mx:VBox>`, and `<mx:HBox>` tags to define Box containers. Use the VBox (vertical box) and HBox (horizontal box) containers as shortcuts so you do not have to specify the `direction` property in the Box container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example creates a Box container with a vertical layout:

```
<?xml version="1.0"?>
<!-- containers\layouts\BoxSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Box direction="vertical"
        borderStyle="solid"
        paddingTop="10"
        paddingBottom="10"
        paddingLeft="10"
        paddingRight="10">

        <mx:Button id="fname" label="Button 1"/>
        <mx:Button id="lname" label="Button 2"/>
        <mx:Button id="addr1" label="Button 3"/>
        <mx:ComboBox id="state"/>
    </mx:Box>
</mx:Application>
```

The following code example is equivalent to the previous example, except that this example defines a vertical Box container by using the `<mx:VBox>` tag:

```
<?xml version="1.0"?>
<!-- containers\layouts\VBoxSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox borderStyle="solid"
        paddingTop="10"
        paddingBottom="10"
        paddingLeft="10"
        paddingRight="10">

        <mx:Button id="fname" label="Button 1"/>
        <mx:Button id="lname" label="Button 2"/>
        <mx:Button id="addr1" label="Button 3"/>
        <mx:ComboBox id="state"/>
    </mx:VBox>
</mx:Application>
```

ControlBar layout container

You use the ControlBar container with a Panel or TitleWindow container to hold components that can be shared by the other children in the Panel or TitleWindow container. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart, as the following example shows:

A ControlBar container has the following default sizing characteristics:

Property	Default value
Default size	The height is the default or explicit height of the tallest child, plus the top and bottom padding of the container. The width is large enough to hold all of its children at the default or explicit width of the children, plus any horizontal gap between the children, plus the left and right padding of the container.
Default padding	10 pixels for the `top`, `bottom`, `left`, and `right` values.

For complete reference information, see ControlBar in *Adobe Flex 2 Language Reference*.

Creating a ControlBar container

You use the `<mx:ControlBar>` tag to define a ControlBar control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML code, either in another tag or in an ActionScript block. You specify the `<mx:ControlBar>` tag as the last child tag of an `<mx:Panel>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\CBarSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function addToCart():void {
                // Handle event.
            }
        ]]>
    </mx:Script>

    <mx:Panel title="My Application"
        paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10">

        <mx:HBox width="250" height="200">
            <!-- Area for your catalog. -->
        </mx:HBox>

        <mx:ControlBar width="250">
            <mx:Label text="Quantity"/>
            <mx:NumericStepper/>
            <!-- Use Spacer to push Button control to the right. -->
            <mx:Spacer width="100%"/>
            <mx:Button label="Add to Cart"
                click="addToCart();"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

ApplicationControlBar layout container

You use the ApplicationControlBar container to hold components that provide access to application navigation elements and commands. An ApplicationControlBar container for an editor, for example, could include Button controls for setting the font weight, a ComboBox to select the font, and a MenuBar control to select the edit mode. The ApplicationControlBar is a sublcass of the ControlBar class; however, it has a different look and feel.

Typically, you place an ApplicationControlBar container at the top of the application, as the following example shows:

If you dock the ApplicationControlBar container at the top of an application, it does not scroll with the application contents.

The ApplicationControlBar container has the following default sizing characteristics:

Property	Default value
Default size	The height is the default or explicit height of the tallest child, plus the top and bottom padding of the container. In normal mode, the width is large enough to hold all of its children at the default or explicit width of the children, plus any horizontal gap between the children, plus the left and right padding of the container. In docked mode, the width equals the application width. If the application is not wide enough to contain all the controls in the ApplicationControlBar container, the bar is clipped.
Default padding	5 pixels for the `top` value. 4 pixels for the `bottom` value. 8 pixels for the `left` and `right` values.

For complete reference information, see ApplicationControlBar in *Adobe Flex 2 Language Reference*.

Creating an ApplicationControlBar container

You use the `<mx:ApplicationControlBar>` tag to define a ControlBar control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML code, either in another tag or in an ActionScript block.

The ApplicationControlBar container can be in either of the following modes:

Docked mode The bar is always at the top of the application's drawing area. Any application-level scroll bars don't apply to the container, so it always remains at the top of the visible area, and the bar expands to fill the width of the application. To created a docked ApplicationControlBar container, set its `dock` property to `true`.

Normal mode The bar can be placed anywhere in the application, is sized and positioned just like any other component, and scrolls with the application. The ApplicationControlBar floats if its `dock` property is `false`. The default value is `false`.

NOTE

In contrast to the ControlBar container, it is possible to set the `backgroundColor` style for an instance of the ApplicationControlBar. The ApplicationControlBar container has two styles, `fillColors` and `fillAlpha`, that are not supported by the ControlBar container.

The following example shows an application with a simple docked ApplicationControlBar that includes a MenuBar. The Application also includes an HBox control that exceeds the application size; when you scroll the application to view the bottom of the HBox control, the ApplicationControlBar control does not scroll.

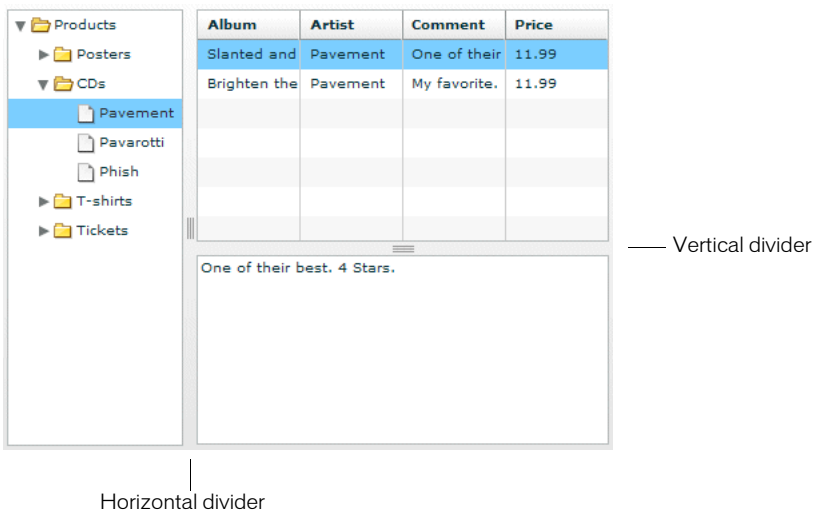
```
<?xml version="1.0"?>
<!-- containers\layouts\AppCBarSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:XMLList id="menuXML">
        <menuitem label="File">
            <menuitem label="New" data="New"/>
            <menuitem label="Open" data="Open"/>
            <menuitem label="Save" data="Save"/>
            <menuitem label="Exit" data="Exit"/>
        </menuitem>
        <menuitem label="Edit">
            <menuitem label="Cut" data="Cut"/>
            <menuitem label="Copy" data="Copy"/>
            <menuitem label="Paste" data="Paste"/>
        </menuitem>
        <menuitem label="View"/>
    </mx:XMLList>

    <mx:Array id="cmbDP">
        <mx:String>Item 1</mx:String>
        <mx:String>Item 2</mx:String>
        <mx:String>Item 3</mx:String>
    </mx:Array>

    <mx:ApplicationControlBar id="dockedBar"
        dock="true">
        <mx:MenuBar height="100%"
            dataProvider="{menuXML}"
            labelField="@label"
            showRoot="true"/>
        <mx:HBox paddingBottom="5"
            paddingTop="5">
            <mx:ComboBox dataProvider="{cmbDP}"/>
            <mx:Spacer width="100%"/>
            <mx:TextInput id="myTI" text=""/>
            <mx:Button id="srch1"
                label="Search"
                click="Alert.show('Searching')"/>
```

```
        </mx:HBox>
    </mx:ApplicationControlBar>

    <mx:TextArea width="300" height="200"/>
</mx:Application>
```

DividedBox, HDividedBox, and VDividedBox layout containers

The DividedBox layout container lays out its children horizontally or vertically, similar to a Box container, except that it inserts a divider between each child. You can use a mouse pointer to move the dividers in order to resize the area of the container allocated to each child. You use the `direction` property of a DividedBox container to determine vertical (default) or horizontal layout. The HDividedBox and VDividedBox containers are DividedBox containers with `horizontal` and `vertical direction` property values.

The following example shows a DividedBox container:



In this example, the outermost container is a horizontal DividedBox container. The horizontal divider marks the border between a Tree control and a vertical DividedBox container.

The vertical DividedBox container holds a DataGrid control (top) and a TextArea control (bottom). The vertical divider marks the border between these two controls.

A DividedBox, HDividedBox, or VDividedBox container has the following default sizing characteristics:

Property	Default value
Default size	**Vertical DividedBox** The height is large enough to hold all of its children at the default or explicit heights of the children, plus any vertical gap between the children, plus the top and bottom padding of the container. The width is the default or explicit width of the widest child, plus the left and right padding of the container. **Horizontal DividedBox** The height is the default or explicit height of the tallest child, plus the top and bottom padding of the container. The width is large enough to hold all of its children at the default or explicit widths of the children, plus any horizontal gap between the children, plus the left and right padding of the container.
Default padding	0 pixels for the `top`, `bottom`, `left`, and `right` values.
Default gap	10 pixels for the horizontal and vertical gaps.

For complete reference information, see DividedBox, HDividedBox, and VDividedBox in *Adobe Flex 2 Language Reference*.

Creating a DividedBox, HDividedBox, or VDividedBox container

You use the `<mx:DividedBox>`, `<mx:VDividedBox>`, and `<mx:HDividedBox>` tags to define DividedBox containers. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. Typically, you use the VDividedBox (vertical DividedBox) and HDividedBox (horizontal DividedBox) containers as shortcuts so that you do not have to specify the `direction` property.

The following code example creates the image shown in "DividedBox, HDividedBox, and VDividedBox layout containers" on page 567:

```
<?xml version="1.0"?>
<!-- containers\layouts\HDivBoxSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="white">

    <mx:Script>
      <![CDATA[
        private function myGrid_initialize():void {
          myGrid.dataProvider = [
            {Artist:'Pavement', Album:'Slanted and Enchanted',
                Price:11.99, Comment:'One of their best. 4 Stars.'},
            {Artist:'Pavement', Album:'Brighten the Corners',
```

```
                Price:11.99, Comment:'My favorite.'}
          ];
        }
      ]]>
    </mx:Script>

    <mx:HDividedBox width="100%" height="100%">
        <mx:Tree id="tree1"
            width="30%" height="100%"
            labelField="@label"
            showRoot="true">
            <mx:XMLList>
                <menuitem label="Products">
                    <menuitem label="Posters" isBranch="true"/>
                    <menuitem label="CDs">
                        <menuitem label="Pavement"/>
                        <menuitem label="Pavarotti"/>
                        <menuitem label="Phish"/>
                    </menuitem>
                    <menuitem label="T-shirts" isBranch="true"/>
                    <menuitem label="Tickets" isBranch="true"/>
                </menuitem>
            </mx:XMLList>
        </mx:Tree>

        <mx:VDividedBox width="70%" height="100%">
            <mx:DataGrid id="myGrid"
                width="100%" height="100%"
                initialize="myGrid_initialize();"
                change="currentMessage.text=
                    event.currentTarget.selectedItem.Comment;"/>
            <mx:TextArea id="currentMessage"
                width="100%"
                height="60"
                text="One of their best. 4 Stars."/>
        </mx:VDividedBox>

    </mx:HDividedBox>
</mx:Application>
```

Notice that this example does not implement the logic to change the top area of the VDividedBox container when you selct a node in the Tree control.

Using the dividers

The dividers of a DividedBox container let you resize the area of the container allocated for a child. However, for the dividers to function, the child has to be resizable, that is, it must specify a percentage-based size. So, a child with an explicit or default height or width cannot be resized in the corresponding direction using a divider. Therefore, when you use the DividedBox container, you typically use percentage sizing for its children to make them resizable.

When you specify a percentage value for the `height` or `width` properties of a child to make it resizable, Flex initially sizes the child to the specified percentage, if possible. Then Flex can resize the child to take up all available space.

You can use the dividers to resize a percentage-sized child up to its maximum size, or down to its minimum size. To constrain the minimum size or maximum size of an area of the DividedBox, set an explicit value for the `minWidth` and `minHeight` properties or the `maxWidth` and `maxHeight` properties of the children in that area.

Using live dragging

By default, the DividedBox container disables live dragging. This means that the DividedBox container does not update the layout of its children until the user finishes dragging the divider, when the user releases the mouse button on a selected divider.

You can configure the DividedBox container to use live dragging by setting the `liveDragging` property to `true`. With live dragging enabled, the DividedBox container updates its layout as the user moves a divider. In some cases, you may encounter decreased performance if you enable live dragging.

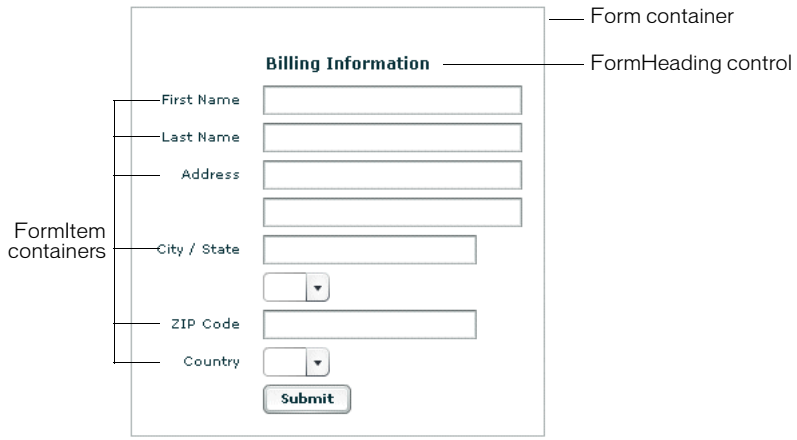
Form, FormHeading, and FormItem layout containers

Forms are one of the most common methods that web applications use to collect information from users. Forms are used for collecting registration, purchase, and billing information, and for many other data collection tasks.

About forms

Flex supports form development by using the Form layout container and several child components of the Form container. The Form container lets you control the layout of a form, mark form fields as required or optional, handle error messages, and bind your form data to the Flex data model to perform data checking and validation. Also, you can apply style sheets to configure the appearance of your forms.

You use three different components to create your forms, as the following example shows:



The following table describes the types of components that you use to create forms in Flex:

Container	Tag	Description
Form	`<mx:Form>`	Defines the container for the entire form, including the overall form layout. Use the FormHeading control and FormItem container to define content. Also, you can insert other types of components into a Form container.
FormHeading	`<mx:FormHeading>`	Defines a heading within your form. You can have multiple FormHeading controls within a single Form container.
FormItem	`<mx:FormItem>`	Contains one or more form children arranged horizontally or vertically. Children can be controls or other containers. A single Form container can hold multiple FormItem containers.

For complete reference information, see Form, FormHeading, and FormItem in *Adobe Flex 2 Language Reference*.

Creating forms

You typically create a form by defining the following elements:

- The Form control
- FormHeading components, nested inside the Form control
- FormItem containers, nested inside the Form control
- Form fields, such as ComboBox and TextInput controls, nested inside the FormItem containers

You can also include other components inside a form, such as HRule controls, as needed.

The following sections discuss how to create the Form, FormHeading, and FormItem components, and present an example of a complete form.

Creating the Form container

The Form container is the outermost container of a Flex form. The primary use of the Form container is to control the sizing and layout of the contents of the form, including the size of labels and the gap between items. The Form container always arranges its children vertically and left-aligns them in the form. The form container contains one or more FormHeading and FormItem containers.

You use the `<mx:Form>` tag to define the Form container. Specify an `id` value if you intend to refer to the entire form elsewhere in your MXML, either in another tag or in an ActionScript block.

The following code example shows the Form container definition for the form shown in the previous image in "About forms" on page 571:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form id="myForm" width="400" height="100">

        <!-- Define FormHeading and FormItem components here -->

    </mx:Form>
</mx:Application>
```

For complete reference information, see Form in *Adobe Flex 2 Language Reference*.

Creating a FormHeading control

A FormHeading control specifies an optional label for a group of FormItem containers. The left side of the label is aligned with the left side of the controls in the form. You can have multiple FormHeading controls in your form to designate multiple content areas. You can also use FormHeading controls with a blank `label` to create vertical space in your form.

You use the `<mx:FormHeading>`, tag to define a FormHeading container. Specify an `id` value if you intend to refer to the heading elsewhere in your MXML, either in another tag or in an ActionScript block.

The following code example defines the FormHeading control for the image shown in "About forms" on page 571:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormHeadingSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form id="myForm" width="400" height="100">

        <mx:FormHeading label="Billing Information"/>

        <!--Define FormItem containers here. -->

    </mx:Form>
</mx:Application>
```

For complete reference information, see FormHeading in *Adobe Flex 2 Language Reference*.

Creating a FormItem container

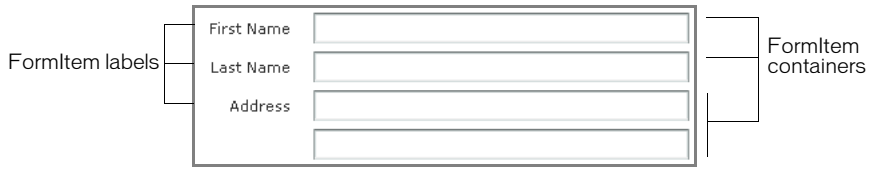
A FormItem container specifies a form element consisting of the following parts:

- A single label
- One or more child controls or containers, such as input controls

The label is vertically aligned with the first child in the FormItem container and is right-aligned in the region to the left of the container.

You use the `<mx:FormItem>`, tag to define a FormHeading container. Specify an `id` value if you intend to refer to the item elsewhere in your MXML, either in another tag or in an ActionScript block.

Form containers typically contain multiple FormItem containers, as the following example shows:



In this example, you define three FormItem containers: one with the label First Name, one with the label Last Name, and one with the label Address. The Address FormItem container holds two controls to let a user enter two lines of address information. Each of the other two FormItem containers includes a single control.

For complete reference information, see FormItem in *Adobe Flex 2 Language Reference*.

Specifying form item direction

When you create a FormItem container, you specify its direction using the value `vertical` (default) or `horizontal` for the `direction` property:

vertical Flex positions children vertically to the right of the FormItem label.

horizontal Flex positions children horizontally to the right of the FormItem label. If all children do not fit on a single row, they are divided into multiple rows with equal-sized columns. You can ensure that all children fit on a single line by using percentage-based widths or by specifying explicit widths wide enough for all of the components.

Controlling form item label style

FormItem control labels are FormItemLabel objects. Therefore, you can control the style of a FormItemLabel by specifying a style for the label by setting a FormItemLabel type style selector. The following example sets the FormItem label color to dark blue:

```
<mx:Style>
  FormItemLabel {
    color: #333399;
  }
</mx:Style>
```

Example: A simple form

The following example shows the FormItem container definitions for the example form:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormComplete.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function setValues():void {
                // Handle value setting.
            }
        ]]>
    </mx:Script>

    <mx:Form id="myForm" width="400">

        <mx:FormHeading label="Billing Information"/>

        <mx:FormItem label="First Name">
            <mx:TextInput id="fname" width="100%"/>
        </mx:FormItem>

        <mx:FormItem label="Last Name">
            <mx:TextInput id="lname" width="100%"/>
        </mx:FormItem>

        <mx:FormItem label="Address">
            <mx:TextInput id="addr1" width="100%"/>
            <mx:TextInput id="addr2" width="100%"/>
        </mx:FormItem>

        <mx:FormItem label="City / State" direction="vertical">
            <mx:TextInput id="city"/>
            <mx:ComboBox id="st" width="50"/>
        </mx:FormItem>

        <mx:FormItem label="Zip Code">
            <mx:TextInput id="zip" width="100"/>
        </mx:FormItem>

        <mx:FormItem label="Country">
            <mx:ComboBox id="cntry"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit"
                click="setValues();"/>
        </mx:FormItem>
    </mx:Form>
```

```
</mx:Application>
```

Laying out forms

Flex determines the default size of a form in the following ways:

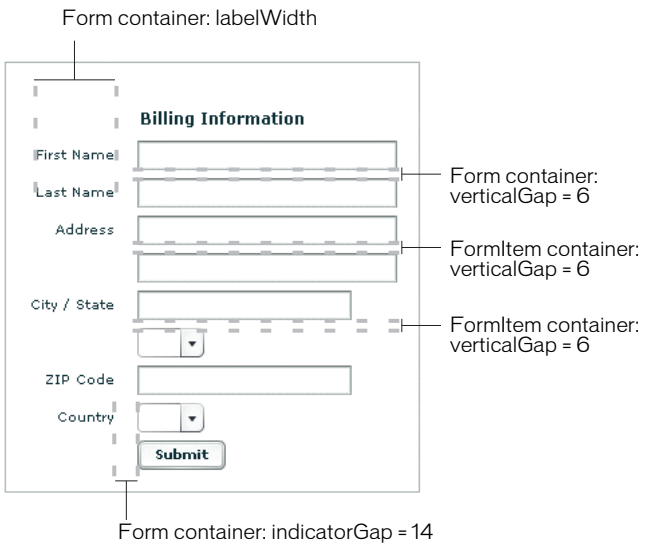
- The default height is large enough to hold the default or explicit heights of all the container children, plus the Form container top and bottom padding and the gaps between children.
- The default width is large enough to accommodate the widest FormItem label, plus the `indicatorGap` between the labels and the child controls, plus the widest default or explicit width among the child controls in the FormItems.

The following sections describe in detail how Form children are aligned and laid out.

Aligning and spacing Form container children

All Form container labels are right-aligned, and all children are left-aligned in the container. You cannot override this alignment.

The following example shows the spacing of Form container children that you can control:



The following table describes the style properties that you use to control spacing and their default values:

Component	Style	Description	Default value
Form	verticalGap	Vertical space between Form container children	6 pixels
	horizontalGap	Horizontal space between Form container children	8 pixels
	labelWidth	Width of labels	Calculated by the container based on the child labels
	paddingTop paddingBottom paddingLeft paddingRight	Border spacing around children	16 pixels on all sides
	indicatorGap	Gap between the end of the area in the form reserved for labels and the FormItem children or FormHeading heading	14 pixels
FormHeading	indicatorGap	Overrides the indicator gap set by the <mx:Form> tag	14 pixels
	paddingTop	Gap between the top of the component and the label text	16 pixels
FormItem	direction	Direction of FormItem children: vertical or horizontal	vertical
	horizontalGap	Horizontal spacing between children in a FormItem container	8 pixels
	labelWidth	The width for the FormItem heading	The width of the label text
	paddingTop paddingBottom paddingLeft paddingRight	Border spacing around the FormItem	0 pixels on all sides
	verticalGap	Vertical spacing between children in a FormItem container	6 pixels
	indicatorGap	Overrides the indicator gap set by the <mx:Form> tag	Determined by the <mx:Form> tag

Sizing and positioning Form container children

The Form layout container arranges children in a vertical column. The area of the Form container that is designated for children does not encompass the entire Form container. Instead, it starts at the right of the area defined by any labels and the gap defined by the `indicatorGap` property. For example, if the width of the Form container is 500 pixels, and the labels and `indicatorGap` property allocate 100 pixels of that width, the width of the child area is 400 pixels.

By default, Flex sizes the Form layout children vertically to their default height. Flex then determines the default width of each child, and stretches the child's width to the next highest quarter of the child area—that is, to one-quarter, one-half, three-quarters, or full width of the child area.

For example, if a container has a child area 400 pixels wide, and the default width of a TextArea control is 125 pixels, Flex stretches the TextArea control horizontally to the next higher quarter of the child area, the 200-pixel boundary, which is one-half of the child area. This sizing algorithm applies only to components without an explicitly specified width. It prevents your containers from having ragged right edges caused by controls with different widths.

You can also explicitly set the height or width of any control in the form to either a pixel value or a percentage of the Form size by using the `height` and `width` properties of the child.

Defining a default button

You use the `defaultButton` property of a container to define a default Button control. Pressing the Enter key while the focus is on any form control activates the Button control just as if it was explicitly selected.

For example, a login form displays user name and password inputs and a submit Button control. Typically, the user types a user name, tabs to the password field, types the password, and presses the Enter key to submit the login information without explicitly selecting the Button control. To define this type of interaction, set the `defaultButton` to the `id` of the submit Button control. In the following example, the event listener for the `click` event of submit button displays an Alert control, to show that Flex triggers this event if the user presses the Enter key when any form field has the focus. The commented-out line in the example would perform the more realistic action of invoking a web service to let the user log in.

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDefButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import flash.events.MouseEvent;
            import mx.controls.Alert;

            private function submitLogin(eventObj:MouseEvent):void {
                // Display an Alert to show the event happened.
                Alert.show("Login Requested");
                // Commented out to work without a web service.
                //myWebService.Login.send();
            }
        ]]>
    </mx:Script>

    <mx:Form defaultButton="{mySubmitButton}">
        <mx:FormItem label="Username">
            <mx:TextInput id="username"
                width="100"/>
        </mx:FormItem>
        <mx:FormItem label="Password">
            <mx:TextInput id="password"
                width="100"
                displayAsPassword="true"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button id="mySubmitButton"
                label="Login"
                click="submitLogin(event);"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

NOTE

When the drop-down list of a ComboBox control is open, pressing Enter selects the currently highlighted item in the ComboBox control; it does not activate the default button.

Specifying required fields

Flex includes support for defining required input fields of a form. To define a required field, you specify the `required` property of the FormItem container. If this property is specified, all the children of the FormItem container are marked as required.

Flex inserts a red asterisk (*) character as a separator between the FormItem label and the FormItem child to indicate a required field. For example, the following example shows an optional ZIP code field and a required ZIP code field:

The following code example defines these fields:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormReqField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form>
        <mx:FormItem label="ZIP Code">
            <mx:TextInput id="zipOptional"
                width="100"/>
        </mx:FormItem>

        <mx:FormItem label="ZIP Code" required="true">
            <mx:TextInput id="zipRequired"
                width="100"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

You can enable the required indicator of a FormItem child at run time. This could be useful when the user input in one form field makes another field required. For example, you might have a form with a CheckBox control that the user selects to subscribe to a newsletter. Checking the box could make the user e-mail field required, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormReqFieldRuntime.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form>
        <mx:FormItem label="Subscribe">
            <mx:CheckBox label="Subscribe?"
                click="emAddr.required=true"/>
        </mx:FormItem>

        <mx:FormItem id="emAddr" label="e-mail address">
            <mx:TextInput id="emailAddr"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Flex does not perform any automatic enforcement of a required field; it only marks fields as required. You must add validation logic to your form to enforce it. As part of your enforcement logic, you can use Flex validators. All Flex validators have a `required` property, which is `true` by default. You can use validators in several ways, depending on how you enforce required fields and validation. For details, see Chapter 40, "Validating Data," on page 1281. For one example of using validators with forms, see "Using a Flex data model to store form data" on page 586.

Storing and validating form data

As part of designing your form, you must consider how you want to store your form data. In Flex, you have the following choices:

- Store the data within the form controls.
- Create a Flex data model to store your data.

Your decision about how to represent your data also affects how you perform input error detection or *data validation*, one of the primary tasks of a robust and stable form. You typically validate user input before you submit the data to the server. You can validate the user input within a submit function, or when a user enters data into the form.

Flex provides a set of data validators for the most common types of data collected by a form. You can use Flex validators with the following types of data:

- Credit card information
- Dates
- E-mail addresses
- Numbers
- Phone numbers
- Social Security Numbers
- Strings
- ZIP codes

As part of building your form, you can perform data validation by using your own custom logic, take advantage of the Flex data validation mechanism, or use a combination of custom logic and Flex data validation.

The following sections include information on how to initiate validation in a form; for detailed information on how to use Flex data validation, see Chapter 40, "Validating Data," on page 1281.

Using Form controls to hold your form data

The following example uses Form controls to store the form data:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormData.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function processValues(zip:String, pn:String):void {
                // Validate and process data.
            }
        ]]>
    </mx:Script>

    <mx:Form id="myForm" defaultButton="{mySubmitButton}">

        <mx:FormItem label="Zip Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit" id="mySubmitButton"
                click="processValues(zipCode.text, phoneNumber.text);"/>
        </mx:FormItem>

    </mx:Form>
</mx:Application>
```

This example form defines two form controls: one for a ZIP code and one for a phone number. When you submit the form, you call a function that takes the two arguments that correspond to the data stored in each control. Your submit function can then perform any data validation on its inputs before processing the form data.

You don't have to pass the data to the submit function. The submit function can access the form control data directly, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataSubmitNoArg.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = zipCode.text;
                var inputPhone:String = phoneNumber.text;
                // Check to see if pn is a number.
                // Check to see if zip is less than 4 digits.
                // Process data.
            }
        ]]>
    </mx:Script>

    <mx:Form id="myForm" defaultButton="{mySubmitButton}">

        <mx:FormItem label="Zip Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit" id="mySubmitButton"
                click="processValues();"/>
        </mx:FormItem>

    </mx:Form>
</mx:Application>
```

The technique of using the form fields directly, however, has the problem that the function is specific to the form and cannot easily be used by other forms.

Validating form control contents data on user entry

To validate form data upon user input, you can add Flex data validators to your application. The following example uses the ZipCodeValidator and PhoneNumbervalidator to perform validation:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataValidate.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = zipCode.text;
                var inputPhone:String = phoneNumber.text;
                // Perform any additional validation.
                // Process data.
            }
        ]]>
    </mx:Script>

    <mx:ZipCodeValidator id="zcVal"
        source="{zipCode}" property="text"
        domain="US or Canada"/>

    <mx:PhoneNumberValidator id="pnVal"
        source="{phoneNumber}" property="text"/>

    <mx:Form id="myForm" defaultButton="{mySubmitButton}">

        <mx:FormItem label="Zip Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>

        <mx:FormItem>
            <mx:Button label="Submit" id="mySubmitButton"
                click="processValues();"/>
        </mx:FormItem>

    </mx:Form>
</mx:Application>
```

If you validate the input data every time the user enters it, you might not have to do so again in your submit function. However, some validation in your submit function might still be necessary, especially if you want to ensure that two fields are valid when compared with each other.

For example, you can use Flex validators to validate a ZIP code field and state field individually. But you might want to validate that the ZIP code is valid for the specified state before submitting the form data. To do so, you perform a second validation in the submit function.

For detailed information on using validators, see Chapter 40, "Validating Data," on page 1281.

Using a Flex data model to store form data

You can use a Flex data model to structure and store your form data and provide a framework for data validation. A data model stores data in fields that represent each part of a specific data set. For example, a *person* model might store information such as a person's name, age, and phone number. You can then validate the data in the model based on the type of data stored in each model field.

The following example defines a Flex data model that contains two values. The two values correspond to the two input fields of a form:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataModel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define the submit function that validates and
        processes the data. -->
    <mx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = myFormModel.zipCodeModel;
                var inputPhone:String = myFormModel.phoneNumberModel;
                // Process data.
            }
        ]]>
    </mx:Script>

    <!-- Define data model. -->
    <mx:Model id="myFormModel">
        <info>
            <zipCodeModel>{zipCode.text}</zipCodeModel>
            <phoneNumberModel>{phoneNumber.text}</phoneNumberModel>
        </info>
    </mx:Model>
```

```
    <!-- Define the form. -->
    <mx:Form borderStyle="solid">
        <mx:FormItem label="Zip Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button id="b1"
                label="Submit"
                click="processValues();"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

You use the `<mx:Model>` tag to define the data model. Each child tag of the data model defines one field of the model. The tag body of each child tag in the model defines a *binding* to a form control. In this example, you bind the `zipCodeModel` model field to the text value of the `zipCode` TextInput control, and you bind the `phoneNumberModel` field to the text value of the `phoneNumber` TextInput control. For more information on data models, see Chapter 39, “Storing Data,” on page 1269.

When you bind a control to a data model, Flex automatically copies data from the control to the model upon user input. In this example, your submit function accesses the data from the model, not directly from the form controls.

Using Flex validators with form models

The following example modifies the example in "Using a Flex data model to store form data" on page 586 by inserting two data validators—one for the ZIP code field and one for the phone number field:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataModel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define the submit function that validates and processes the data -
->
    <mx:Script>
        <![CDATA[
            private function processValues():void {
                var inputZip:String = myFormModel.zipCodeModel;
                var inputPhone:String = myFormModel.phoneNumberModel;
                // Process data.
            }
        ]]>
    </mx:Script>

    <!-- Define data model. -->
    <mx:Model id="myFormModel">
        <info>
            <zipCodeModel>{zipCode.text}</zipCodeModel>
            <phoneNumberModel>{phoneNumber.text}</phoneNumberModel>
        </info>
    </mx:Model>

    <!-- Define validators. -->
    <mx:ZipCodeValidator
        source="{myFormModel}" property="zipCodeModel"
        trigger="{zipCode}"
        listener="{zipCode}"/>
    <mx:PhoneNumberValidator
        source="{myFormModel}" property="phoneNumberModel"
        trigger="{b1}"
        listener="{phoneNumber}"
        triggerEvent="click"/>

    <!-- Define the form. -->
    <mx:Form borderStyle="solid">
        <mx:FormItem label="Zip Code">
            <mx:TextInput id="zipCode"/>
        </mx:FormItem>
        <mx:FormItem label="Phone Number">
            <mx:TextInput id="phoneNumber"/>
        </mx:FormItem>
        <mx:FormItem>
```

```
            <mx:Button id="b1"
                label="Submit"
                click="processValues();"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

When the user enters data into the `zipCode` form field, Flex automatically copies that data to the data model. The `ZipCodeValidator` validator gets invoked when the user exits the `zipCode` form field, as specified by the validator's `trigger` property and the default value of the `triggerEvent` property, `valueCommit`. Flex then draws a red box around the `zipCode` field, as specified by the `listener` property.

When the user enters data into the `phoneNumber` form field, Flex automatically copies that data to the data model. The `PhoneNumberValidator` validator gets invoked when the user clicks the Button control, as specified by the validator's `trigger` and `triggerEvent` properties. Flex then draws a red box around the `phoneNumber` field, as specified by the `listener` property.

For detailed information on using validators, see Chapter 40, "Validating Data," on page 1281.

Populating a Form control from a data model

Another use for data models is to include data in the model to populate form fields with values. The following example shows a form that reads static data from a data model to obtain the value for a form field:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataFromModel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define data model. -->
    <mx:Model id="myFormModel">
        <info>
            <fName>{firstName.text}</fName>
            <lName>{lastName.text}</lName>
            <department>Accounting</department>
        </info>
    </mx:Model>

    <mx:Form>
        <mx:FormItem label="First and Last Names">
            <mx:TextInput id="firstName"/>
            <mx:TextInput id="lastName"/>
        </mx:FormItem>
        <mx:FormItem label="Department">
            <mx:TextInput id="dept" text="{myFormModel.department}"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

This department data is considered static because the form always shows the same value in the field. You could also create a dynamic data model that takes the value of the department field from a web service, or calculates it based on user input.

For more information on data models, see Chapter 39, “Storing Data,” on page 1269.

Submitting data to a server

Form data is typically processed on a server, not locally on the client. Therefore, the submit event listener must have a mechanism for packing the form data for transfer to the server, and then handling any results returned from the server. In Flex, you typically use a web service, HTTP service, or remote Java object to pass data to the server.

You can also build logic into your submit function to control navigation of your application when the submit succeeds and when it fails. When the submit succeeds, you typically navigate to an area of your application that displays the results. If the submit fails, you can return control to the form so that the user can fix any errors.

The following example adds a web service to process form input data. In this example, the user enters a ZIP code, and then selects the Submit button. After performing any data validation, the submit event listener calls the web service to obtain the city name, current temperature, and forecast for the ZIP code.

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataSubmitServer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define the web service connection.
        The specified WSDL URI is not functional. -->
    <mx:WebService id="WeatherService"
        wsdl="/ws/WeatherService?wsdl">
        <mx:operation name="GetWeather">
            <mx:request>
                <ZipCode>{zipCode.text}</ZipCode>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:Script>
        <![CDATA[
            private function processValues():void {
                // Check to see if ZIP code is valid.
                WeatherService.GetWeather.send();
            }
        ]]>
    </mx:Script>

    <mx:Form>
        <mx:FormItem label="Zip Code">
            <mx:TextInput id="zipCode"
                width="200"
                text="Zip code please?"/>
            <mx:Button
                width="60"
                label="Submit"
                click="processValues();"/>
        </mx:FormItem>
    </mx:Form>

    <mx:VBox>
        <mx:TextArea
            text=
              "{WeatherService.GetWeather.lastResult.CityShortName}"/>
        <mx:TextArea
            text=
              "{WeatherService.GetWeather.lastResult.CurrentTemp}"/>
        <mx:TextArea
            text=
```

```
            "{WeatherService.GetWeather.lastResult.DayForecast}"/>
    </mx:VBox>
</mx:Application>
```

This example binds the form's input `zipCode` field directly to the `ZipCode` field of the web service request. To display the results from the web service, you bind the results to controls in a VBox container.

You have a great deal of flexibility when passing data to a web service. For example, you might modify this example to bind the input form field to a data model, and then bind the data model to the web service request. For more information on using web services, see Chapter 45, "Using RPC Components," on page 1407.

You can also add event listeners for the web service to handle both a successful call to the web service, using the `result` event, and a call that generates an error, using the `fault` event. An error condition might cause you to display a message to the user with a description of the error. For a successful result, you might navigate to another section of your application.

The following example adds a `load` event and a `fault` event to the form. In this example, the form is defined as one child of a ViewStack container, and the form results are defined as a second child of the ViewStack container:

```
<?xml version="1.0"?>
<!-- containers\layouts\FormDataSubmitServerEvents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define the web service connection.
        The specified WSDL URI is not functional. -->
    <mx:WebService id="WeatherService"
        wsdl="/ws/WeatherService?wsdl"
        result="successfulCall();"
        fault="errorCall();">
        <mx:operation name="GetWeather">
            <mx:request>
                <ZipCode>{zipCode.text}</ZipCode>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;

            private function processValues():void {
                // Check to see if ZIP code is valid.
                WeatherService.GetWeather.send();
            }

            private function successfulCall():void {
```

```
                vs1.selectedIndex=1;
            }

            private function errorCall():void {
                Alert.show("Web service failed!", "Alert Box", Alert.OK);
            }
        ]]>
    </mx:Script>

    <mx:ViewStack id="vs1">
        <mx:Form>
            <mx:FormItem label="Zip Code">
                <mx:TextInput id="zipCode"
                    width="200"
                    text="Zip code please?"/>
                <mx:Button width="60"
                    label="Submit"
                    click="processValues();"/>
            </mx:FormItem>
        </mx:Form>

        <mx:VBox>
            <mx:TextArea
                text=
                   "{WeatherService.GetWeather.lastResult.CityShortName}"/>
            <mx:TextArea
                text=
                    "{WeatherService.GetWeather.lastResult.CurrentTemp}"/>
            <mx:TextArea
                text=
                    "{WeatherService.GetWeather.lastResult.DayForecast}"/>
        </mx:VBox>
    </mx:ViewStack>
</mx:Application>
```

When a call to the web service succeeds, the `successfulCall()` function switches the current ViewStack child to the VBox container to show the returned results. An error from the web service displays an Alert box, but does not change the current child of the ViewStack container; the form remains visible, which lets the user fix any input errors.

You have many options for handling navigation in your application based on the results of the submit function. The previous example used a ViewStack container to handle navigation. You might also choose to use a TabNavigator container or Accordion container for this same purpose.

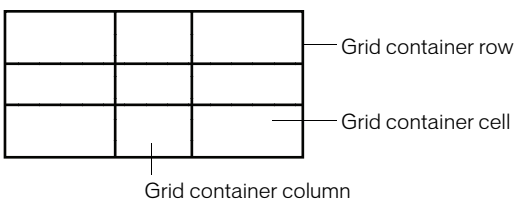
In some applications, you might choose to embed the form in a TitleWindow container. A TitleWindow container is a pop-up window that appears above the Adobe Flash Player drawing surface. In this scenario, users enter form data and submit the form from the TitleWindow container. If a submit succeeds, the TitleWindow container closes and displays the results in another area of your application. If a submit fails, Flex displays an error message and leaves the TitleWindow container visible.

Another type of application might use a dashboard layout, where you have multiple panels open on the dashboard. Submitting the form could cause another area of the dashboard to update with results, while a failure could display an error message.

For more information on the TabNavigator, Accordion, and TitleWindow containers, see Chapter 16, "Using Navigator Containers," on page 627.

Grid layout container

You use a Grid layout container to arrange children as rows and columns of cells, much like an HTML table. The following example shows a Grid container that consists of nine cells arranged in a three-by-three pattern:



You can put zero or one child in each cell of a Grid container. If you include multiple children in a cell, put a container in the cell, and then add children to the container. The height of all cells in a row is the same, but each row can have a different height. The width of all cells in a column is the same, but each column can have a different width.

You can define a different number of cells for each row or each column of the Grid container. In addition, a cell can span multiple columns and/or multiple rows of the container.

The Grid, GridRow, and GridItem containers have the following default sizing characteristics:

Property	Default value
Grid height	The sum of the default or explicit heights of all rows plus the gaps between rows.
Grid width	The sum of the default or explicit width of all columns plus the gaps between columns.

Property	Default value
Height of each row and each cell	The default or explicit height of the tallest item in the row. If a GridItem container does not have an explicit size, its default height is the default or explicit height of the child in the cell.
Width of each column and each cell	The default or explicit width of the widest item in the column. If a GridItem container not have an explicit width, its default width is the default or explicit width of the child in the cell.
Gap between rows and columns	Determined by the `horizontalGap` and `verticalGap` properties of the Grid class. The default value for both gaps is 6 pixels.
Default padding	0 pixels for the `top`, `bottom`, `left`, and `right` values, for all three container classes.

If the default or explicit size of the child is larger than the size of an explicitly sized cell, the child is clipped at the cell boundaries.

If the child's default width or default height is smaller than the cell, the default horizontal alignment of the child in the cell is `left` and the default vertical alignment is `top`. You can use the `horizontalAlign` and `verticalAlign` properties of the `<mx:GridItem>` tag to control positioning of the child.

For complete reference information, see Grid, GridRow, and GridItem in *Adobe Flex 2 Language Reference*. For information on the Tile container, which creates a layout where all cells have the same size, see "Tile layout container" on page 606.

Creating a Grid layout container

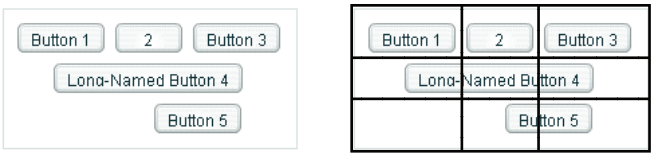
You create a Grid layout container as follows:

- You use the `<mx:Grid>` tag to define a Grid container; it can hold any number of `<mx:GridRow>` child tags.
- You use the `<mx:GridRow>` tag to define each row. It must be a child of the `<mx:Grid>` tag and can hold any number of `<mx:GridItem>` child tags.
- You use the `<mx:GridItem>` tag to define each row cell; it must be a child of the `<mx:GridRow>` tag.

The <mx:GridItem> tag takes the following optional properties that control how the item is laid out:

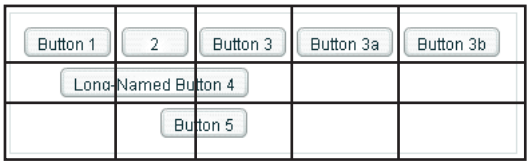
Property	Type	Use	Descriptions
rowSpan	Number	Property	Specifies the number of rows of the Grid container spanned by the cell. The default value is 1. You cannot extend a cell past the number of rows in the Grid container.
colSpan	Number	Property	Specifies the number of columns of the Grid container spanned by the cell. The default value is 1. You cannot extend a cell past the number of columns in the Grid container.

The following image shows a Grid container with three rows and three columns:



On the left, you see how the Grid container appears in Flash Player. On the right, you see the Grid container with borders overlaying it to illustrate the configuration of the rows and columns. In this example, the buttons in the first (top) row each occupy a single cell. The button in the second row spans three columns, and the button in the third row spans the second and third columns.

You do not have to define the same number of cells for every row of a Grid container, as the following image shows. The Grid container defines five cells in row one; row two has one item that spans three cells; and row 3 has one empty cell, followed by an item that spans two cells.



The following MXML code creates the Grid container with three rows and three columns shown in the first image in this section:

```
<?xml version="1.0"?>
<!-- containers\layouts\GridSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Grid id="myGrid">

        <!-- Define Row 1. -->
        <mx:GridRow id="row1">
            <!-- Define the first cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 1"/>
            </mx:GridItem>
            <!-- Define the second cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="2"/>
            </mx:GridItem>
            <!-- Define the third cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 3"/>
            </mx:GridItem>
        </mx:GridRow>

        <!-- Define Row 2. -->
        <mx:GridRow id="row2">
            <!-- Define a single cell to span three columns of Row 2. -->
            <mx:GridItem colSpan="3" horizontalAlign="center">
                <mx:Button label="Long-Named Button 4"/>
            </mx:GridItem>
        </mx:GridRow>

        <!-- Define Row 3. -->
        <mx:GridRow id="row3">
            <!-- Define an empty first cell of Row 3. -->
            <mx:GridItem/>
            <!-- Define a cell to span columns 2 and 3 of Row 3. -->
            <mx:GridItem colSpan="2" horizontalAlign="center">
                <mx:Button label="Button 5"/>
            </mx:GridItem>
        </mx:GridRow>

    </mx:Grid>
</mx:Application>
```

To modify the preceding example to include five buttons across the top row, you modify the first <mx:GridRow> tag as follows:

```
<?xml version="1.0"?>
<!-- containers\layouts\Grid5Button.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Grid id="myGrid">

        <!-- Define Row 1. -->
        <mx:GridRow id="row1">
            <!-- Define the first cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 1"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="2"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3a"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3b"/>
            </mx:GridItem>
        </mx:GridRow>

        <!-- Define Row 2. -->
        <mx:GridRow id="row2">
            <!-- Define a single cell to span three columns of Row 2. -->
            <mx:GridItem colSpan="3" horizontalAlign="center">
                <mx:Button label="Long-Named Button 4"/>
            </mx:GridItem>
        </mx:GridRow>

        <!-- Define Row 3. -->
        <mx:GridRow id="row3">
            <!-- Define an empty first cell of Row 3. -->
            <mx:GridItem/>
            <!-- Define a cell to span columns 2 and 3 of Row 3. -->
            <mx:GridItem colSpan="2" horizontalAlign="center">
                <mx:Button label="Button 5"/>
            </mx:GridItem>
        </mx:GridRow>

    </mx:Grid>
</mx:Application>
```

Setting the row and column span

The `colSpan` and `rowSpan` properties of the GridItem container let you create grid items that occupy multiple grid rows and columns. Making an item span multiple rows or columns does not necessarily make its child control or container larger; you must size the child so it fits the space appropriately, as the following example shows.

The following image shows a modification to the example in "Creating a Grid layout container", where Button 3a now spans two rows, Button 3b spans three rows, and Button 5 spans three columns:

The following code shows the changes that were made to produce these results:

```
<?xml version="1.0"?>
<!-- containers\layouts\GridRowSpan.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Grid id="myGrid">

        <!-- Define Row 1. -->
        <mx:GridRow id="row1" height="33%">
            <!-- Define the first cell of Row 1. -->
            <mx:GridItem>
                <mx:Button label="Button 1"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="2"/>
            </mx:GridItem>
            <mx:GridItem>
                <mx:Button label="Button 3"/>
            </mx:GridItem>
            <mx:GridItem rowSpan="2">
                <mx:Button label="Button 3a" height="100%"/>
            </mx:GridItem>
            <mx:GridItem rowSpan="3">
                <mx:Button label="Button 3b" height="100%"/>
            </mx:GridItem>
        </mx:GridRow>

        <!-- Define Row 2. -->
        <mx:GridRow id="row2" height="33%">
            <!-- Define a single cell to span three columns of Row 2. -->
            <mx:GridItem colSpan="3" horizontalAlign="center">
                <mx:Button label="Long-Named Button 4"/>
            </mx:GridItem>
        </mx:GridRow>

        <!-- Define Row 3. -->
        <mx:GridRow id="row3" height="33%">
            <!-- Define an empty first cell of Row 3. -->
            <mx:GridItem/>
            <!-- Define a cell to span columns 2 and 3 and 4 of Row 3. -->
            <mx:GridItem colSpan="3">
                <mx:Button
                    label="Button 5 expands across 3 columns"
                    width="75%"/>
            </mx:GridItem>
        </mx:GridRow>

    </mx:Grid>
</mx:Application>
```

This example makes several changes with the following effects:

- It sets the height of each row to 33% of the grid, ensuring that the rows have equal heights.
- It sets the `rowSpan` properties of the items with Buttons 3a and 3b to make them span two and three rows, respectively.
- It sets the `height` properties of Buttons 3a and 3b to 100% to make these buttons fill all rows that they span. If you omit this property on these buttons, Flex sets the buttons to their default height, so they do not appear to span the rows.
- It sets Button 5 to span three rows and sets a percentage-based width of 75%. In this example, the text requires the button to fill the available width of the three columns, so Flex sizes the button to the default size that fits the text, not the requested 75%. If you omit the `width` property, the result is identical. To see the effect of the percentage width specification, keep the specification and shorten the label text; the button then spans three-quarters of the three columns, centered on the middle column.

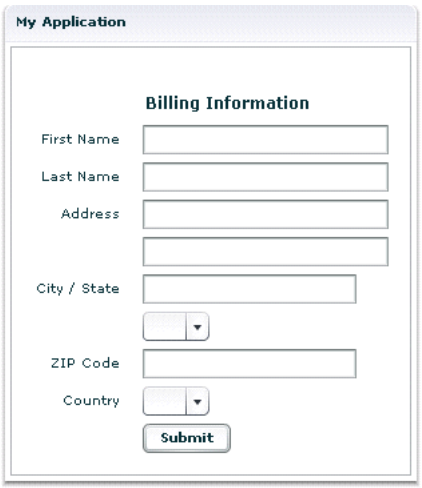
The resulting grid has the several additional characteristics. Although the second row definition specifies only a single `<mx:GridItem>` tag that defines a cell spanning three columns, Flex automatically creates two additional cells to allow Buttons 3a and 3b to expand into the row. The third row also has five cells. The first cell is defined by the empty `<mx:gridItem/>` tag. The second through fourth cells are defined by the `GridItem` that contains Button 5, which spans three columns. The fifth column is created because the last item in the first row spans all three rows.

Panel layout container

A Panel layout container includes a title bar, a title, a status message area (in the title bar), a border, and a content area for its children. Typically, you use Panel containers to wrap self-contained application modules. For example, you could define several Panel containers in your application where one Panel container holds a form, a second holds a shopping cart, and a third holds a catalog.

The Panel container has a `layout` property that lets you specify one of three types of layout: `horizontal`, `vertical` (default), or `absolute` layout. Horizontal and vertical layout use the Flex automatic layout rules to lay out children horizontally or vertically. Absolute layout requires you to specify each child's *x* and *y* position relative to the panel contents area, or to use constraint-based layout styles to anchor one or more sides or the container horizontal or vertical center relative to the panel content area. For examples of using absolute and constraint-based layout in a container, see "Creating and using a Canvas control" on page 555. For detailed information on using these layout techniques, see Chapter 8, "Sizing and Positioning Components," on page 221.

The following example shows a Panel container with a Form container as its child:



The Panel container has the following default sizing characteristics:

Property	Default value
Default size	The height is large enough to hold all of its children at the default height of the children, plus any vertical gaps between the children, the top and bottom padding, the top and bottom borders, and the title bar. The width is the larger of the default width of the widest child plus the left and right padding of the container, or the width of the title text, plus the border.
Padding	4 pixels for the `top`, `bottom`, `left`, and `right` values.

For complete reference information, see Panel in *Adobe Flex 2 Language Reference*.

Creating a Panel layout container

You define a Panel container in MXML by using the `<mx:Panel>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example defines a Panel container that contains a form as the top-level container in your application. In this example, the Panel container provides you with a mechanism for including a title bar, as in a standard GUI window.

```
<?xml version="1.0"?>
<!-- containers\layouts\PanelSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Panel id="myPanel" title="My Application">

        <mx:Form width="300">
            <mx:FormHeading label="Billing Information"/>
            <!-- Form contents goes here -->
        </mx:Form>

    </mx:Panel>
</mx:Application>
```

Adding a ControlBar container to a Panel container

You can use the ControlBar container with a Panel container to hold components that can be shared by the other children in the Panel container. The RichTextEditor control, for example, consists of a Panel control with a TextArea control and a custom ControlBar for the text formatting controls. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart, as the following example shows:

You specify the <mx:ControlBar> tag as the last child tag of an <mx:Panel> tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\PanelCBar.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Panel title="My Application"
        paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10"
        width="300">

        <mx:Script>
            <![CDATA[
                private function addToCart():void {
                    // Handle event.
                }
            ]]>
        </mx:Script>

        <mx:HBox width="100%">
            <!-- Area for your catalog. -->
        </mx:HBox>

        <mx:ControlBar width="100%">
            <mx:Label text="Quantity"/>
            <mx:NumericStepper id="myNS"/>
            <!-- Use Spacer to push Button control to the right. -->
            <mx:Spacer width="100%"/>
            <mx:Button label="Add to Cart" click="addToCart();"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

For more information on the ControlBar container, see "ControlBar layout container" on page 562.

You can also add the ControlBar container dynamically to a Panel container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\PanelCBarDynamicAdd.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.containers.ControlBar;
            import mx.controls.*;
            import flash.events.MouseEvent;

            private var myCB:ControlBar=new ControlBar();
            private var myLabel:Label=new Label();
            private var myNS:NumericStepper=new NumericStepper();
            private var mySpacer:Spacer=new Spacer();
            private var myButton:Button=new Button();

            private function addCBHandler():void {
                // Create Controlbar control.
                myLabel.text="Quantity";
                mySpacer.percentWidth=100;
                myButton.label="Add to Cart";
                myButton.addEventListener('click', addToCart);


                myCB.percentWidth=100;
                myCB.addChild(myLabel);
                myCB.addChild(myNS);
                myCB.addChild(mySpacer);
                myCB.addChild(myButton);

                // Add the ControlBar as the last child of the
                // Panel container.
                // The ControlBar appears in the normal content area
                // of the Panel container.
                myPanel.addChildAt(myCB, myPanel.numChildren);
                // Call createComponentsFromDescriptors() to make the
                // ControlBar appear in the bottom border area
                // of the Panel container. The ControlBar must be the
                // last child in the Panel container.
                myPanel.createComponentsFromDescriptors();
            }

            private function addToCart(event:MouseEvent):void {
                Alert.show("ControlBar Button clicked.");
            }
        ]]>
```

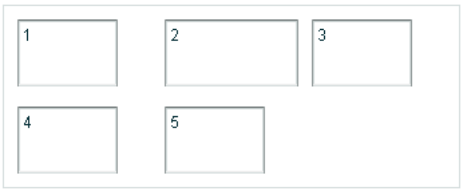
```
    </mx:Script>

    <mx:Panel id="myPanel"
        title="My Application"
        paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10"
        width="300">

        <mx:HBox width="100%">
            <!-- Area for your catalog. -->
        </mx:HBox>

        <mx:Button label="Add ControlBar" click="addCBHandler();"/>

    </mx:Panel>
</mx:Application>
```

Tile layout container

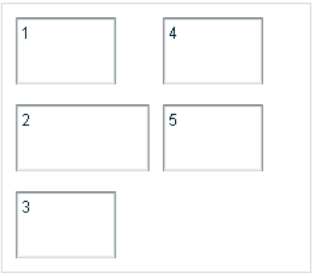
A Tile layout container lays out its children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The `direction` property determines the layout. The valid values for the `direction` property are `vertical` for a column layout and `horizontal` (default) for a row layout.

All Tile container cells have the same size, unlike the cells of a Grid layout container (see "Grid layout container" on page 594). Flex arranges the cells of a Tile container in a square grid, where each cell holds a single child component. For example, if you define 16 children in a Tile layout container, Flex lays it out four cells wide and four cells high. If you define 13 children, Flex still lays it out four cells wide and four cells high, but leaves the last three cells in the fourth row empty.

The following image shows examples of horizontal and vertical Tile containers:



Horizontal (default)



Vertical

A Tile container has the following default sizing characteristics:

Property	Default value
`direction`	`horizontal`
Default size of all cells	The height is the default or explicit height of the tallest child; width is the default or explicit width of the widest child. All cells have the same default size.
Default size of Tile container	Flex computes the square root of the number of children, and rounds up to the nearest integer. For example, if there are 26 children, the square root is 5.1, which is rounded up to 6. Flex then lays out the Tile container in a 6 by 6 grid. The default height of the Tile container is equal to (tile cell default height) * (rounded square root of the number of children), plus any gaps between children and any padding. The default width is equal to (tile cell default width) * (rounded square root of the number of children), any gaps between children and any padding.
Minimum size of Tile container	The default size of a single cell. Flex always allocates enough space to display at least one cell.
Default padding	0 pixels for the `top`, `bottom`, `left`, and `right` values.

For complete reference information, see Tile in *Adobe Flex 2 Language Reference*.

Creating a Tile layout container

You define a Tile container in MXML by using the `<mx:Tile>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. You can use the `tileHeight` and `tileWidth` properties to specify explicit tile dimensions.

The following example creates the horizontal Tile container shown in the image in "Tile layout container" on page 606. All cells have the height and width of the largest child: 50 pixels high and 100 pixels wide.

```
<?xml version="1.0"?>
<!-- containers\layouts\TileSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Tile id="myFlow"
        direction="horizontal"
        borderStyle="solid"
        paddingTop="10" paddingBottom="10"
        paddingRight="10" paddingLeft="10"
        verticalGap="15" horizontalGap="10">

        <mx:TextInput id="text1" text="1" height="50" width="75"/>
        <mx:TextInput id="text2" text="2" height="50" width="100"/>
        <mx:TextInput id="text3" text="3" height="50" width="75"/>
        <mx:TextInput id="text4" text="4" height="50" width="75"/>
        <mx:TextInput id="text5" text="5" height="50" width="75"/>
    </mx:Tile>
</mx:Application>
```

Sizing and positioning a child in a Tile container

Flex sets the default size of each Tile cell to the height of the tallest child and the width of the widest child. All cells have the same default size. If the default size of a child is larger than the cell because, for example, you used the `tileHeight` and `tileWidth` properties to explicitly size the cells, Flex automatically sizes the child to fit inside the cell boundaries. This, in turn, may clip the content inside the control; for instance, the label of a button might be clipped even though the button itself fits into the cell. If you specify an explicit child dimension that is greater than the `tileHeight` or `tileWidth` property, Flex clips the child.

If the child's default width or default height is smaller than the cell, the default horizontal alignment of the child in the cell is `left` and the default vertical alignment is `top`. You can use the `horizontalAlign` and `verticalAlign` properties of the `<mx:Tile>` tag to control the positioning of the child.

If the child uses percentage-based sizing, the child is enlarged or shrunk to the specified percentage of the cell. In the example in "Creating a Tile layout container" on page 607, the TextInput control named `text2` has a width of 100 pixels; therefore, the default width of all Tile cells is 100 pixels, so most children are smaller than the cell size. If you want all child controls to increase in size to the full width of the cells, set the `width` property of each child to 100%, as the following example shows. The example also shows how to use the Tile control's `tileWidth` property to specify the width of the tiles:

```
<?xml version="1.0"?>
<!-- containers\layouts\TileSizing.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Tile id="myFlow"
        direction="horizontal"
        borderStyle="solid"
        paddingTop="10" paddingBottom="10"
        paddingRight="10" paddingLeft="10"
        verticalGap="15" horizontalGap="10"
        tileWidth="100">

        <mx:TextInput id="fname" text="1" height="50" width="100%"/>
        <mx:TextInput id="lname" text="2" height="50" width="100%"/>
        <mx:TextInput id="addr1" text="3" height="50" width="100%"/>
        <mx:TextInput id="addr2" text="4" height="50" width="100%"/>
        <mx:TextInput id="addr3" text="5" height="50" width="100%"/>
    </mx:Tile>
</mx:Application>
```

NOTE

When you set the child's `width` and `height` properties to percentages, the percentage is based on the size of the tile cell, not on the size of the Tile container itself. Even though it isn't an explicitly defined container, the cell acts as the parent of the components in the Tile container.

TitleWindow layout container

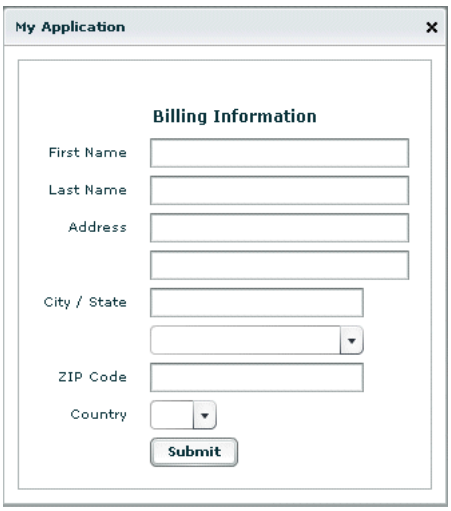
A TitleWindow layout container is a Panel container that is optimized for use as a pop-up window. The container consists of a title bar, a caption and status area in the title bar, a border, and a content area for its children. Unlike the Panel container, it can display a Close button, and is designed to work as a pop-up window that users can drag around the screen the application window.

A pop-up TitleWindow container can be *modal*, which means that it takes all keyboard and mouse input until it is closed, or *nonmodal*, which means other windows can accept input while the pop-up window is still open.

One typical use for a TitleWindow container is to hold a form. When the user completes the form, you can close the TitleWindow container programmatically, or let the user request the application to close it by using the close icon (a box with an *x* inside it) in the upper-right corner of the window.

Because you pop up a Title window, you do not create it directly in MXML, as you do most controls; instead you use the PopUpManager.

The following example shows a TitleWindow container with a Form container as its child:



The TitleWindow container has the following default sizing characteristics:

Property	Default value
Default size	The height is large enough to hold all of the children in the content are at the default or explicit heights of the children, plus the title bar and border, plus any vertical gap between the children, plus the top and bottom padding of the container. The width is the larger of the default or explicit width of the widest child, plus the left and right container borders padding, or the width of the title text.
`borders`	10 pixels for the `left` and `right` values. 2 pixels for the `top` value. 0 pixels for the `bottom` value.
`padding`	4 pixels for the `top`, `bottom`, `left`, and `right` values.

For complete reference information, see TitleWindow in *Adobe Flex 2 Language Reference*.

Using the PopUpManager to create a TitleWindow container

To create and remove a pop-up TitleWindow container, you use methods of the PopUpManager. The PopUpManager is in the mx.managers package.

Creating a pop-up window

To create a pop-up window, use the PopUpManager's `createPopUp()` method. The `createPopUp()` method has the following signature:

```
public static createPopUp(parent:DisplayObject, class:Class,
  modal:Boolean = false) : IFlexDisplayObject
```

The method has the following arguments:

Argument	Description
parent	A reference to a window to pop-up over.
class	A reference to the class of object you want to create, typically a custom MXML component that implements a TitleWindow container.
modal	(Optional) A Boolean value that indicates whether the window is modal, and takes all mouse input until it is closed (`true`), or whether interaction is allowed with other controls while the window is displayed (`false`). The default value is `false`.

NOTE

Flex continues executing code in the parent even after you create a modal pop-up window.

You can also create a pop-up window by passing an instance of a TitleWindow class or custom component to the PopUpManager's `addPopUp()` method. For more information, see "Using the addPopUp() method" on page 624.

Defining a custom TitleWindow component

One of the most common ways of creating a TitleWindow container is to define it as a custom MXML component.

- You define the TitleWindow container, its event handlers, and all of its children in the custom component.
- You use the PopUpManager's `createPopUp()` and `removePopUp()` methods to create and remove the TitleWindow container.

The following example code defines a custom MyLoginForm TitleWindow component that is used as a pop-up window:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\MyLoginForm.mxml -->
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;

            private function processLogin():void {
                // Check credentials (not shown) then remove pop up.
                PopUpManager.removePopUp(this);
            }
        ]]>
    </mx:Script>

    <mx:Form>
        <mx:FormItem label="User Name">
            <mx:TextInput id="username" width="100%"/>
        </mx:FormItem>
        <mx:FormItem label="Password">
            <mx:TextInput id="password"
                displayAsPassword="true"
                width="100%"/>
        </mx:FormItem>
    </mx:Form>
    <mx:HBox>
        <mx:Button click="processLogin();" label="OK"/>
        <mx:Button
            label="Cancel"
            click="PopUpManager.removePopUp(this);"/>
    </mx:HBox>
</mx:TitleWindow>
```

This file, named MyLoginForm.mxml, defines a TitleWindow container by using the `<mx:TitleWindow>` tag. The TitleWindow container defines two TextInput controls, for user name and password, and two Button controls, for submitting the form and for closing the TitleWindow container. This example does not include the code for verifying the user name and password in the `submitForm()` event listener.

In this example, you process the form data in an event listener of the MyLoginForm.mxml component. To make this component more reusable, you can define the event listeners in your main application. This lets you create a generic form that leaves the data handling to the application that uses the form. For an example that defines the event listeners in the main application, see "Using TitleWindow and PopUpManager events" on page 616.

Using the PopUpManager to create the pop-up TitleWindow

To create a pop-up window, you call the PopUpManager's `createPopUp()` method and pass it the parent, the name of the class that creates the pop-up, and the modal Boolean value. The following main application code creates the TitleWindow container defined in "Defining a custom TitleWindow component":

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginForm.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import mx.core.IFlexDisplayObject;
            import myComponents.MyLoginForm;

            private function showLogin():void {
                // Create a non-modal TitleWindow container.
                var helpWindow:IFlexDisplayObject =
                    PopUpManager.createPopUp(this, MyLoginForm, false);
            }
        ]]>
    </mx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
    </mx:VBox>
</mx:Application>
```

In this example, when the user selects the Login button, the event listener for the `click` event uses the `createPopUp()` method to create a TitleWindow container, passing to it the name of the MyLoginForm.mxml file as the class name.

Often, you cast the return value of the `createPopUp()` method to TitleWindow so that you can manipulate the properties of the pop-up TitleWindow container, as the following version of the `showLogin()` method from the preceding example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginFormCast.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
          import mx.managers.PopUpManager;
          import mx.core.IFlexDisplayObject;
          import myComponents.MyLoginForm;

          // Additional import statement to use the TitleWindow container.
          import mx.containers.TitleWindow;

          private function showLogin():void {
            // Create the TitleWindow container.
            var helpWindow:TitleWindow =
          TitleWindow(PopUpManager.createPopUp(this, MyLoginForm, false));

            // Add title to the title bar.
            helpWindow.title="Enter Login Information";

            // Make title bar slightly transparent.
            helpWindow.setStyle("borderAlpha", 0.9);

            // Add a close button.
           // To close the container, your must also handle the close event.
            helpWindow.showCloseButton=true;
          }
        ]]>
    </mx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
    </mx:VBox>
</mx:Application>
```

Removing a pop-up window

To remove a pop-up TitleWindow, use the PopUpManager's `removePopUp()` method. You pass the object created with the `PopUpManager.createPopUp()` method to the `removePopUp()` method. The following modification to the example from "Using the PopUpManager to create the pop-up TitleWindow" on page 613 removes the pop-up when the user clicks the Done button:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginFormRemove.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
          import mx.managers.PopUpManager;
          import myComponents.MyLoginForm;
          import mx.core.IFlexDisplayObject;

          private var helpWindow:IFlexDisplayObject;

          private function showLogin():void {
            // Create the TitleWindow container.
           helpWindow = PopUpManager.createPopUp(this, MyLoginForm, false);
          }

          private function removeForm():void {
            PopUpManager.removePopUp(helpWindow);
          }
        ]]>
    </mx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
        <mx:Button id="b2" label="Remove Form" click="removeForm();"/>
    </mx:VBox>
</mx:Application>
```

You commonly call the `removePopUp()` method from a TitleWindow's `close` event and the PopUpManager `mouseDownOutside` event as described in "Passing data using events" on page 622.

Using TitleWindow and PopUpManager events

You can add a close icon (a small *x* in the upper-right corner of the TitleWindow's title bar) to make it appear similar to dialog boxes in a GUI environment. You do this by setting the `showCloseButton` property to `true`, as the following example shows:

```
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
  showCloseButton="true">
```

The default value for `showCloseButton` is `false`.

The TitleWindow broadcasts a `close` event when the user clicks the close icon. You must create a handler for that event and close the TitleWindow from within that event handler. Flex does not close the window automatically.

In the simplest case, you can call the PopUpManager `removePopUp()` method directly in the TitleWindow `close` event property specify, as the following line shows:

```
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
  showCloseButton="true"
  close="PopUpManager.removePopUp(this);">
```

If you need to clean up, before closing the TitleWindow control, create an event listener function for the close event and close the TitleWindow from within that handler, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\MyLoginFormRemoveMe.mxml -->
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
    showCloseButton="true"
    close="removeMe();">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;

            private function removeMe():void {
                // Put any clean-up code here.
                PopUpManager.removePopUp(this);
            }
        ]]>
    </mx:Script>

</mx:TitleWindow>
```

You can also use the PopUpManager `mouseDownOutside` event to close the pop-up window when a user clicks the mouse outside the TitleWindow. To do this, you add an event listener to the TitleWindow instance for the `mouseDownOutside` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainMyLoginFormRemove.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import mx.containers.TitleWindow;
            import myComponents.MyLoginForm;
            import mx.events.FlexMouseEvent;

            private var helpWindow:TitleWindow;

            private function showLogin():void {
                // Create the TitleWindow container.
                helpWindow = TitleWindow(PopUpManager.createPopUp(this,
                    MyLoginForm, false));
               helpWindow.addEventListener("mouseDownOutside", removeForm);
            }

            private function removeForm(event:FlexMouseEvent):void {
                PopUpManager.removePopUp(helpWindow);
            }
        ]]>
    </mx:Script>

    <mx:VBox width="300" height="300">
        <mx:Button click="showLogin();" label="Login"/>
    </mx:VBox>
</mx:Application>
```

You define the event listener in the main application, and then assign it to the pop-up window when you create it. This technique lets you use a generic pop-up window, defined by the component MyLoginForm.mxml, and then modify the behavior of the component by assigning event listeners to it from the main application.

Centering a pop-up window

Call the `centerPopUp()` method of the PopUpManager to center the pop-up within another container. The following custom MXML component centers itself in its parent container:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\MyLoginFormCenter.mxml -->
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="handleCreationComplete();">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;

            private function handleCreationComplete():void {
                // Center the TitleWindow container
                // over the control that created it.
                PopUpManager.centerPopUp(this);
            }

            private function processLogin():void {
                // Check credentials (not shown) then remove pop up.
                PopUpManager.removePopUp(this);
            }
        ]]>
    </mx:Script>
    <mx:Form>
        <mx:FormItem label="User Name">
            <mx:TextInput id="username" width="100%"/>
        </mx:FormItem>
        <mx:FormItem label="Password">
            <mx:TextInput id="password"
                width="100%"
                displayAsPassword="true"/>
        </mx:FormItem>
    </mx:Form>
    <mx:HBox>
        <mx:Button click="processLogin();" label="OK"/>
        <mx:Button
            label="Cancel"
            click="PopUpManager.removePopUp(this);"/>
    </mx:HBox>
</mx:TitleWindow>
```

Creating a modal pop-up window

The `createPopUp()` method takes an optional *modal* parameter. You can set this parameter to `true` to make the window modal. When a TitleWindow is modal, you cannot select any other component while the window is open. The default value of *modal* is `false`.

The following example creates a modal pop-up window:

```
var pop1:IFlexDisplayObject = PopUpManager.createPopUp(this, MyLoginForm,
  true);
```

Passing data to and from a pop-up window

To make your pop-up window more flexible, you might want to pass data to it or return data from it. To do this, use the following guidelines:

- Create a custom component to be your pop-up. In most circumstances, this component is a TitleWindow container.
- Declare variables in your pop-up that you will set in the application that creates the pop-up.
- Cast the pop-up to be the same type as your custom component.
- Pass a reference to that component to the pop-up window, if the pop-up window is to set a value on the application or one of the application's components.

For example, the following application populates a ComboBox in the pop-up window with an Array defined in the main application.

When creating the pop-up, the application casts the pop-up to be of type ArrayEntryForm, which is the name of the custom component that defines the pop-up window. If you do not do this, the application cannot access the properties that you create.

The application passes a reference to the TextInput component in the Application container to the pop-up window so that the pop-up can write its results back to the container. The application also passes the Array of file extensions for the pop-up ComboBox control's data provider, and sets the pop-up window's title. By setting these in the application, you can reuse the pop-up window in other parts of the application without modification, because it does not have to know the name of the component it is writing back to or the data that it is displaying, only that its data is in an Array and it is writing to a TextArea.

```
<?xml version="1.0"?>
<!-- containers\layouts\MainArrayEntryForm.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import myComponents.ArrayEntryForm;

            public var helpWindow:Object;

            public function displayForm():void {
                // Array with data for the custom control ComboBox control.
                var doctypes:Array = ["*.as", "*.mxml", "*.swc"]

                // Create the pop-up and cast the
                // return value of the createPopUp()
                // method to the ArrayEntryForm custom component.
                var pop1:ArrayEntryForm = ArrayEntryForm(
                    PopUpManager.createPopUp(this, ArrayEntryForm, true));
                // Set TitleWindow properties.
                pop1.title="Select File Type";
                pop1.showCloseButton=true;

                // Set properties of the ArrayEntryForm custom component.
                pop1.targetComponent = ti1;
                pop1.myArray = doctypes;
                PopUpManager.centerPopUp(pop1);
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <mx:TextInput id="ti1" text=""/>
    </mx:VBox>
    <mx:Button id="b1" label="Select File Type" click="displayForm();"/>
</mx:Application>
```

The following custom component, ArrayEntryForm.mxml, declares two variables. The first one is for the Array that the parent application passes to the pop-up window. The second holds a reference to the parent application's TextInput control. The component uses that reference to update the parent application:

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\ArrayEntryForm.mxml -->
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
        showCloseButton="true"
        width="200" borderAlpha="1"
        close="removeMe();">

    <mx:Script>
        <![CDATA[
            import mx.controls.TextInput;
            import mx.managers.PopUpManager;

            // Variables whose values are set by the main application.
            // Data provider array for the component's ComboBox control.
            [Bindable]
            public var myArray:Array;
            // A reference to the TextInput control
            // in which to put the result.
            public var targetComponent:TextInput;

            // OK button click event listener.
            // Sets the target component in the application to the
            // selected ComboBox item value.
            private function submitData():void {
                targetComponent.text = String(cb1.selectedItem);
                removeMe();
            }

            // Cancel button click event listener.
            private function removeMe():void {
                PopUpManager.removePopUp(this);
            }
        ]]>
    </mx:Script>

    <mx:ComboBox id="cb1" dataProvider="{myArray}"/>
    <mx:HBox>
        <mx:Button label="OK" click="submitData();"/>
        <mx:Button label="Cancel" click="removeMe();"/>
    </mx:HBox>
</mx:TitleWindow>
```

From within a pop-up custom component, you can also access properties of the parent application using the `parentApplication` property. For example, if the application has a Button control named b1, you can get the label of that Button control, as the following example shows:

```
myLabel = parentApplication.b1.label;
```

This technique, however, uses a hard-coded value in the pop-up component for both the target component id in the parent and the property in the component.

Passing data using events

The following example modifies the example from the previous section to use event listeners defined in the main application to handle the passing of data back from the pop-up window to the main application. This example shows the ArrayEntryFormEvents.mxml file with no event listeners defined within it.

```
<?xml version="1.0"?>
<!-- containers\layouts\myComponents\ArrayEntryFormEvents.mxml -->
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
    showCloseButton="true"
    width="200"
    borderAlpha="1">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;

            // Variables whose values are set by the main application.
            // Data provider array for the component's ComboBox control.
            [Bindable]
            public var myArray:Array;

        ]]>
    </mx:Script>

    <mx:ComboBox id="cb1" dataProvider="{myArray}"/>
    <mx:HBox>
        <mx:Button id="okButton" label="OK"/>
        <mx:Button id="cancelButton" label="Cancel"/>
    </mx:HBox>
</mx:TitleWindow>
```

The main application defines the event listeners and registers them with the controls defined within the pop-up window:

```
<?xml version="1.0"?>
<!-- containers\layouts\MainArrayEntryFormEvents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import flash.events.Event;
            import myComponents.ArrayEntryFormEvents;

            public var pop1:ArrayEntryFormEvents;

            public function displayForm():void {
                // Array with data for the custom control ComboBox control.
                var doctypes:Array = ["*.as", "*.mxml", "*.swc"]

                // Create the pop-up and cast the return value
                // of the createPopUp()
                // method to the ArrayEntryFormEvents custom component.
                pop1 = ArrayEntryFormEvents(
              PopUpManager.createPopUp(this, ArrayEntryFormEvents, true));
                // Set TitleWindow properties.
                pop1.title="Select File Type";
                pop1.showCloseButton=true;

                // Set the event listeners for
                // the ArrayEntryFormEvents component.
                pop1.addEventListener("close", removeMe);
               pop1["cancelButton"].addEventListener("click", removeMe);
                pop1["okButton"].addEventListener("click", submitData);

              // Set properties of the ArrayEntryFormEvents custom control.
                pop1.myArray = doctypes;
                PopUpManager.centerPopUp(pop1);
            }

            // OK button click event listener.
            // Sets the target component in the application to the
            // selected ComboBox item value.
            private function submitData(event:Event):void {
                ti1.text = String(pop1.cb1.selectedItem);
                removeMe(event);
            }

            // Cancel button click event listener.
            private function removeMe(event:Event):void {
                PopUpManager.removePopUp(pop1);
```

```
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <mx:TextInput id="ti1" text=""/>
    </mx:VBox>
    <mx:Button id="b1" label="Select File Type" click="displayForm();"/>
</mx:Application>
```

Using the addPopUp() method

You can use the `addPopUp()` method of the PopUpManager to create a pop-up window without defining a custom component. This method takes an instance of any class that implements IFlexDisplayObject. Because it takes a class instance, not a class, you can use ActionScript code in an `<mx:Script>` block to create the component instance to pop up, rather than as a separate custom component.

Using the `addPopUp()` method may be preferable to using the `createPopUp()` method if you have to pop up a simple dialog box that is never reused elsewhere. However, it is not the best coding practice if the pop-up is complex or cannot be reused elsewhere.

The following example creates a pop-up with `addPopUp()` method and adds a Button control to that window that closes the window when you click it:

```
<?xml version="1.0"?>
<!-- containers\layouts\MyPopUpButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="600" width="600" >

    <mx:Script>
        <![CDATA[
            import mx.containers.TitleWindow;
            import flash.events.*;
            import mx.managers.PopUpManager;
            import mx.controls.Button;
            import mx.core.IFlexDisplayObject;

            // The variable for the TitleWindow container
            public var myTitleWindow:TitleWindow = new TitleWindow();

            // Method to instantiate and display a TitleWindow container.
            // This is the initial Button control's click event handler.
            public function openWindow(event:MouseEvent):void {
                // Set the TitleWindow container properties.
                myTitleWindow = new TitleWindow();
                myTitleWindow.title = "My Window Title";
                myTitleWindow.width= 220;
```

```
                myTitleWindow.height= 150;
                // Call the method to add the Button control to the
                // TitleWindow container.
                populateWindow();
               // Use the PopUpManager to display the TitleWindow container.
                PopUpManager.addPopUp(myTitleWindow, this, true);
            }

            // The method to create and add the Button child control to the
            // TitleWindow container.
            public function populateWindow():void {
                var btn1:Button = new Button();
                btn1.label="close";
                btn1.addEventListener(MouseEvent.CLICK, closeTitleWindow);
                myTitleWindow.addChild(btn1);
            }

            // The method to close the TitleWindow container.
            public function closeTitleWindow(event:MouseEvent):void {
                PopUpManager.removePopUp(event.currentTarget.parent);
            }
        ]]>
    </mx:Script>
    <mx:Button label="Open Window" click="openWindow(event);"/>
</mx:Application>
```

You can make any component pop up. The following example pops up a TextArea control. The example resizes the control, and listens for a Shift-click to determine when to close the TextArea. Whatever text you type in the TextArea is stored in a Label control in the application when the pop-up window is removed.

```
<?xml version="1.0"?>
<!-- containers\layouts\MyPopUpTextArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            import mx.controls.TextArea;
            import mx.core.IFlexDisplayObject;

            public var myPopUp:TextArea

            public function openWindow(event:MouseEvent):void {
                myPopUp = new TextArea();
                myPopUp.width= 220;
                myPopUp.height= 150;
                myPopUp.text = "Hold down the Shift key, and " +
                    "click in the TextArea to close it.";
                myPopUp.addEventListener(MouseEvent.CLICK, closeWindow);
                PopUpManager.addPopUp(myPopUp, this, true);
                PopUpManager.centerPopUp(myPopUp);
            }

            public function closeWindow(event:MouseEvent):void {
                if (event.shiftKey) {
                    label1.text = myPopUp.text;
        PopUpManager.removePopUp(IFlexDisplayObject(event.currentTarget));
                }
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <mx:Button id="b1" label="Create TextArea Popup"
            click="openWindow(event);"/>
        <mx:Label id="label1"/>
    </mx:VBox>
</mx:Application>
```

CHAPTER 16

Using Navigator Containers

16

Navigator containers control user movement, or navigation, among multiple children where the children are other containers. The individual child containers of the navigator container oversee the layout and positioning of their children; the navigator container does not oversee layout and positioning.

This topic describes navigator containers and their syntax, and provides examples of using navigator containers.

Contents

About navigator containers

A navigator container controls user movement through a group of child containers. For example, a TabNavigator container lets you select the visible child container by using a set of tabs.

NOTE

The direct children of a navigator container must be containers, either layout or navigator containers. You cannot directly nest a control within a navigator; controls must be children of a child container of the navigator container.

Flex provides the following navigator containers:

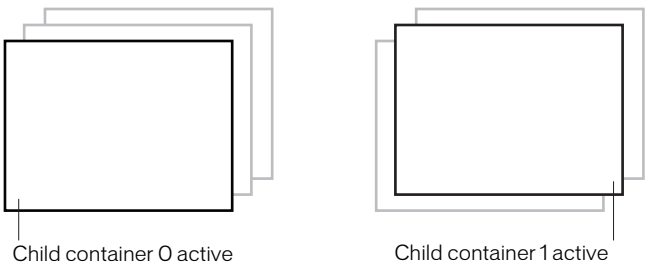
- ViewStack
- TabNavigator
- Accordion

The following sections describe how to use each of the Flex navigator containers.

ViewStack navigator container

A ViewStack navigator container is made up of a collection of child containers that are *stacked* on top of each other, with only one container visible, or *active*, at a time. The ViewStack container does not define a built-in mechanism for users to switch the currently active container; you must use a LinkBar, TabBar, ButtonBar, or ToggleButtonBar control or build the logic yourself in ActionScript to let users change the currently active child. For example, you can define a set of Button controls that switch among the child containers.

The following image shows the stacked child containers in a ViewStack container:



On the left, you see a ViewStack container with the first child active. The index of a child in a ViewStack container is numbered from 0 to *n* - 1, where *n* is the number of child containers. The container on the right is a ViewStack container with the second child container active.

The ViewStack container has the following default sizing characteristics:

Property	Default value
Default size	The width and height of the initially active child.
Container resizing rules	ViewStack containers are sized only once to fit the size of the first child container by default. They do not resize when you navigate to other child containers by default. To force ViewStack containers to resize when you navigate to a different child container, set the `resizeToContent` property to `true`.
Child sizing rules	Children are sized to their default size. The child is clipped if it is larger than the ViewStack container. If the child is smaller than the ViewStack container, it is aligned to the upper-left corner of the ViewStack container.
Default padding	0 pixels for `top`, `bottom`, `left`, and `right` values.

For complete reference information, see ViewStack in *Adobe Flex 2 Language Reference*.

Creating a ViewStack container

You use the `< mx:ViewStack>` tag to define a ViewStack container, and you define the child containers in the tag body. You use the following properties of the ViewStack container to control the active child container:

- `selectedIndex` The index of the currently active container if one or more child containers are defined. The value of this property is -1 if no child containers are defined. The index is numbered from 0 to `numChildren` - 1, where `numChildren` is the number of child containers in the ViewStack container. Set this property to the index of the container that you want active.

  You can use the `selectedIndex` property of the `<mx:ViewStack>` tag to set the default active container when your application starts. The following example sets the index of the default active container to 1:

  ```
  <mx:ViewStack id="myViewStack" selectedIndex="1">
  ```

  The following example uses ActionScript to set the `selectedIndex` property so that the active child container is the second container in the stack:

  ```
  myViewStack.selectedIndex=1;
  ```

- `selectedChild` The currently active container if one or more child containers are defined. The value of this property is `null` if no child containers are defined. Set this property in ActionScript to the identifier of the container that you want active.

  You can set this property only in an ActionScript statement, not in MXML.

  The following example uses ActionScript to set the `selectedChild` property so that the active child container is the child container with an identifier of `search`:

  ```
  myViewStack.selectedChild=search;
  ```

- `numChildren` Contains the number of child containers in the ViewStack container.

  The following example uses the `numChildren` property in an ActionScript statement to set the active child container to the last container in the stack:

  ```
  myViewStack.selectedIndex=myViewStack.numChildren-1;
  ```

NOTE

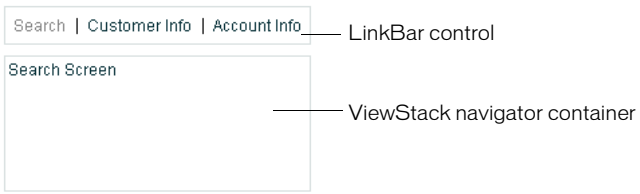
The default creation policy for all containers, except the Application container, is the policy of the parent container. The default policy for the Application container is auto. In most cases, therefore, the View Stack control's children are not created until they are selected. You cannot set the `selectedChild` property to a child that is not yet created.

The following example creates a ViewStack container with three child containers. The example also defines three Button controls that when clicked, select the active child container:

```
<?xml version="1.0"?>
<!-- containers\navigators\VSSimple.mxml  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Create a VBox container so the container for
        the buttons appears above the ViewStack container. -->
    <mx:VBox>
        <!-- Create an HBox container to hold the three buttons. -->
        <mx:HBox borderStyle="solid">

            <!-- Define the three buttons.
                Each uses the child container identifier
                to set the active child container. -->
            <mx:Button id="searchButton"
                label="Search Screen"
                click="myViewStack.selectedChild=search;"/>

            <mx:Button id="cInfoButton"
                label="Customer Info Screen"
                click="myViewStack.selectedChild=custInfo;"/>

            <mx:Button id="aInfoButton"
                label="Account Info Screen"
                click="myViewStack.selectedChild=accountInfo;"/>
        </mx:HBox>

       <!-- Define the ViewStack and the three child containers and have it
            resize up to the size of the container for the buttons. -->
        <mx:ViewStack id="myViewStack"
            borderStyle="solid" width="100%">

            <mx:Canvas id="search" label="Search">
                <mx:Label text="Search Screen"/>
            </mx:Canvas>

            <mx:Canvas id="custInfo" label="Customer Info">
                <mx:Label text="Customer Info"/>
            </mx:Canvas>

            <mx:Canvas id="accountInfo" label="Account Info">
                <mx:Label text="Account Info"/>
            </mx:Canvas>
        </mx:ViewStack>
    </mx:VBox>
</mx:Application>
```

When this example loads, the three Button controls appear, and the first child container defined in the ViewStack container is active. Select a Button control to change the active container.

You can also use a LinkBar, TabBar, ButtonBar, or ToggleButtonBar control to set the active child of a ViewStack container. These classes are subclasses of the NavBar class, so they are collectively called navigator bar controls. Navigator bar controls can determine the number of child containers in a ViewStack container, and then create a horizontal or vertical set of links, tabs, or buttons that let the user select the active child, as the following image shows:



The items in the LinkBar control correspond to the values of the `label` property of each child of the ViewStack container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\navigators\VSLinkBar.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <!-- Create a LinkBar control to navigate
            the ViewStack container. -->
        <mx:LinkBar dataProvider="{myViewStack}" borderStyle="solid"/>

        <!-- Define the ViewStack and the three child containers. -->
        <mx:ViewStack id="myViewStack"
            borderStyle="solid"
            width="100%">

            <mx:Canvas id="search" label="Search">
                <mx:Label text="Search Screen"/>
            </mx:Canvas>

            <mx:Canvas id="custInfo" label="Customer Info">
                <mx:Label text="Customer Info"/>
            </mx:Canvas>

            <mx:Canvas id="accountInfo" label="Account Info">
                <mx:Label text="Account Info"/>
            </mx:Canvas>
        </mx:ViewStack>
    </mx:VBox>
</mx:Application>
```

You provide only a single property to the navigator bar control, `dataProvider`, to specify the name of the ViewStack container associated with it. For more information on the LinkBar control, see "LinkBar control" on page 280. For more information on the TabBar control, see "TabBar control" on page 283. For more information on the ButtonBar and ToggleButtonBar controls, see "ButtonBar and ToggleButtonBar controls" on page 276.

Sizing the children of a ViewStack container

The default width and height of a ViewStack container is the width and height of the first child. A ViewStack container does not change size every time you change the active child.

You can use the following techniques to control the size of a ViewStack container so that it displays all the components inside its children:

- Set explicit `width` and `height` properties for all children to the same fixed values.
- Set percentage-based `width` and `height` properties for all children to the same fixed values.
- Set `width` and `height` properties for the ViewStack container to a fixed or percentage-based value.

The technique that you use is based on your application and the content of your ViewStack container.

Applying behaviors to a ViewStack container

You can assign effects to the ViewStack container or to its children. For example, if you assign the WipeRight effect to the `creationCompleteEffect` property of the ViewStack control, Flex plays the effect once when the ViewStack first appears.

To specify the effects that run when the ViewStack changes children, use the children's `hideEffect` and `showEffect` properties. The effect specified by the `hideEffect` property plays as a container is being hidden, and the effect specified by the `showEffect` property plays as the newly visible child appears. The ViewStack container waits for the completion of the effect specified by the `hideEffect` property for the container that is being hidden before it reveals the new child container.

The following example runs a WipeRight effect when the ViewStack container is first created, and runs a WipeDown effect when each child is hidden and a WipeUp effect when a new child appears.

```
<?xml version="1.0"?>
<!-- containers\navigators\VSLinkEffects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:WipeUp id="myWU" duration="300"/>
    <mx:WipeDown id="myWD" duration="300"/>
    <mx:WipeRight id="myWR" duration="300"/>

    <mx:VBox>
        <mx:LinkBar dataProvider="{myViewStack}"
            borderStyle="solid"
            backgroundColor="#EEEEFF"/>

        <mx:ViewStack id="myViewStack"
            borderStyle="solid"
            width="100%"
            creationCompleteEffect="{myWR}">

            <mx:Canvas id="search"
                label="Search"
                hideEffect="{myWD}"
                showEffect="{myWU}">
                <mx:Label text="Search Screen"/>
            </mx:Canvas>

            <mx:Canvas id="custInfo"
                label="Customer Info"
                hideEffect="{myWD}"
                showEffect="{myWU}">
                <mx:Label text="Customer Info"/>
            </mx:Canvas>

            <mx:Canvas id="accountInfo"
                label="Account Info"
                hideEffect="{myWD}"
                showEffect="{myWU}">
                <mx:Label text="Account Info"/>
            </mx:Canvas>
        </mx:ViewStack>
    </mx:VBox>
</mx:Application>
```

Notice that the `showEffect` property of a container is only triggered when the container's visibility changes from `false` to `true`. Therefore, you use the `creationCompleteEffect` property to trigger the effect when Flex first creates the component.

TabNavigator container

A TabNavigator container creates and manages a set of tabs, which you use to navigate among its children. The children of a TabNavigator container are other containers. The TabNavigator container creates one tab for each child. When the user selects a tab, the TabNavigator container displays the associated child, as the following image shows:

The TabNavigator container is a child class of the ViewStack container and inherits much of its functionality. A TabNavigator container has the following default sizing characteristics:

Property	Default value
Default size	The default or explicit width and height of the first active child plus the tabs, at their default or explicit heights and widths. The default tab height is determined by the font, style, and skin applied to the TabNavigator container.
Container resizing rules	By default, TabNavigator containers are only sized once to fit the size of the first child container by default. They do not resize when you navigate to other child containers by default. To force TabNavigator containers to resize when you navigate to a different child container, set the `resizeToContent` property to `true`.
Child layout rules	The child is clipped if it is larger than the TabNavigator container. If the child is smaller than the TabNavigator container, it is aligned to the upper-left corner of the TabNavigator container.
Default padding	0 pixels for the `top`, `bottom`, `left`, and `right` values.

For complete reference information, see TabNavigator in *Adobe Flex 2 Language Reference*.

Creating a TabNavigator container

You use the `<mx:TabNavigator>` tag to define a TabNavigator container. Only one child of the TabNavigator container is visible at a time. Users can make any child the selected child by selecting its associated tab or by using keyboard navigation controls. Whenever the user changes the current child, the TabNavigator container broadcasts a `change` event.

The TabNavigator container automatically creates a tab for each of its children and determines the tab text from the `label` property of the child, and the tab icon from the `icon` property of the child. The tabs are arranged left to right in the order determined by the child indexes. All tabs are visible, unless they do not fit within the width of the TabNavigator container.

If you disable a child of a TabNavigator container by setting its `enabled` property to `false`, you also disable the associated tab.

The following code creates the TabNavigator container in the image in "TabNavigator container" on page 634:

```
<?xml version="1.0"?>
<!-- containers\navigators\TNSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TabNavigator borderStyle="solid" >

        <mx:VBox label="Accounts"
            width="300"
            height="150">
            <!-- Accounts view goes here. -->
        </mx:VBox>

        <mx:VBox label="Stocks"
            width="300"
            height="150">
            <!-- Stocks view goes here. -->
        </mx:VBox>

        <mx:VBox label="Futures"
            width="300"
            height="150">
            <!-- Futures view goes here. -->
        </mx:VBox>

    </mx:TabNavigator>
</mx:Application>
```

You can also set the currently active child using the `selectedChild` and `selectedIndex` properties inherited from the ViewStack container as follows:

- `numChildren` The index of the currently active container if one or more child containers are defined. The index is numbered from 0 to `numChildren` - 1, where `numChildren` is the number of child containers in the TabNavigator container. Set this property to the index of the container that you want active.
- `selectedChild` The currently active container, if one or more child containers are defined. This property is `null` if no child containers are defined. Set this property to the identifier of the container that you want active. You can set this property only in an ActionScript statement, not in MXML.

For more information on the `selectedChild` and `selectedIndex` properties, including examples, see "ViewStack navigator container" on page 628.

You use the `showEffect` and `hideEffect` properties of the children of a TabNavigator container to specify an effect to play when the user changes the currently active child. The following example plays the WipeLeft effect each time the selected tab changes:

```
<?xml version="1.0"?>
<!-- containers\navigators\TNEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:WipeLeft id="myWL"/>

    <mx:TabNavigator>

        <mx:VBox label="Accounts"
            width="300"
            height="150"
            showEffect="{myWL}">
            <!-- Accounts view goes here. -->
        </mx:VBox>

        <mx:VBox label="Stocks"
            width="300"
            height="150"
            showEffect="{myWL}">
            <!-- Stocks view goes here. -->
        </mx:VBox>

        <mx:VBox label="Futures"
            width="300"
            height="150"
            showEffect="{myWL}">
            <!-- Futures view goes here. -->
        </mx:VBox>

    </mx:TabNavigator>
</mx:Application>
```

Sizing the children of a TabNavigator container

The default width and height of a TabNavigator container is the width and height of the first child. A TabNavigator container does not change size every time you change the active child.

You can use the following techniques to control the size of a TabNavigator container so that it displays all the components inside its children:

- Set explicit `width` and `height` properties for all children to the same fixed values.
- Set percentage-based `width` and `height` properties for all children to the same fixed values.

- Set explicit or percentage-based `width` and `height` properties for the TabNavigator container.

The method that you use is based on your application and the content of your TabNavigator container.

TabNavigator container Keyboard navigation

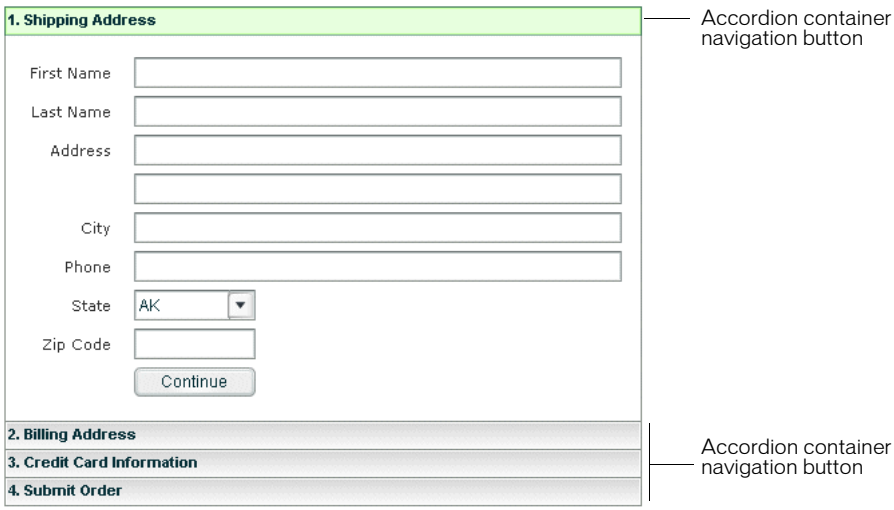
When a TabNavigator container has focus, Flex processes keystrokes as the following table describes:

Key	Action
Down Arrow Right Arrow	Gives focus to the next tab, wrapping from last to first, without changing the selected child.
Up Arrow Left Arrow	Gives focus to the previous tab, wrapping from first to last, without changing the selected child.
Page Down	Selects the next child, wrapping from last to first.
Page Up	Selects the previous child, wrapping from first to last.
Home	Selects the first child.
End	Selects the last child.
Enter Spacebar	Selects the child associated with the tab displaying focus.

Accordion navigator container

Forms are a basic component of many applications. However, users have difficulty navigating through complex forms, or moving back and forth through multipage forms. Sometimes, forms can be so large that they do not fit onto a single screen.

Flex includes the Accordion navigator container, which can greatly improve the look and navigation of a form. The Accordion container defines a sequence of child panels, but displays only one panel at a time. The following image shows an example of an Accordion container:



To navigate a container, the user clicks the navigation button that corresponds to the child panel that they want to access. Accordion containers let users access the child panels in any order to move back and forth through the form. For example, when the user is in the Credit Card Information panel, they might decide to change the information in the Billing Address panel. To do so, they navigate to the Billing Address panel, edit the information, and then navigate back to the Credit Card Information panel.

In HTML, a form that contains shipping address, billing address, and credit card information is often implemented as three separate pages, which requires the user to submit each page to the server before moving on to the next. An Accordion container can organize the information on three child panels with a single Submit button. This architecture minimizes server traffic and lets the user maintain a better sense of progress and context.

NOTE	An empty Accordion container with no child panels cannot take focus.

Although Accordion containers are useful for working with forms and Form containers, you can use any Flex component within a child panel of an Accordion container. For example, you could create a catalog of products in an Accordion container, where each panel contains a group of similar products.

An Accordion container has the following default sizing characteristics:

Property	Default value
Default size	The width and height of the currently active child.
Container resizing rules	Accordion containers are only sized once to fit the size of the first child container by default. They do not resize when you navigate to other child containers by default. To force Accordion containers to resize when you navigate to a different child container, set the `resizeToContent` property to `true`.
Child sizing rules	Children are sized to their default size. The child is clipped if it is larger than the Accordion container. If the child is smaller than the Accordion container, it is aligned to the upper-left corner of the Accordion container.
Default padding	-1 pixel for the `top`, `bottom`, `left`, and `right` values.

For complete reference information, see Accordion in *Adobe Flex 2 Language Reference*.

Creating an Accordion container

You use the <mx:Accordion> tag to define an Accordion container. In the Accordion container, you define one container for each child panel. For example, if the Accordion container has four child panels that the correspond to four parts of a form, you define each child panel by using the Form container, as the following example shows:

```
<?xml version="1.0"?>
<!-- containers\navigators\AccordionSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Accordion id="accordion1" height="450">

        <mx:Form id="shippingAddress" label="1. Shipping Address">

            <mx:FormItem id="sfirstNameItem" label="First Name">
                <mx:TextInput id="sfirstName"/>
            </mx:FormItem>

            <!-- Additional contents goes here. -->

        </mx:Form>

        <mx:Form id="billingAddress" label="2. Billing Address">
            <!-- Form contents goes here. -->
        </mx:Form>

        <mx:Form id="creditCardInfo" label="3. Credit Card Information">
            <!-- Form contents goes here. -->
        </mx:Form>

        <mx:Form id="submitOrder"   label="4. Submit Order">
            <!-- Form contents goes here. -->
        </mx:Form>

    </mx:Accordion>
</mx:Application>
```

This example defines each child panel by using a Form container. However, you can use any container to define a child panel.

NOTE You can use any container to define child panels. However, some containers do not belong in child panels, such as a TabNavigator container because it has no child components, or an Accordion container.

Accordion container Keyboard navigation

When an Accordion container has focus, Flex processes keystrokes as the following table describes:

Key	Action
Down Arrow Right Arrow	Gives focus to the next button, wrapping from last to first, without changing the selected child.
Up Arrow Left Arrow	Gives focus to the previous button, wrapping from first to last, without changing the selected child.
Page Up	Moves to the previous child panel, wrapping from first to last.
Page Down	Moves to the next child panel, wrapping from last to first.
Home	Moves to the first child panel.
End	Moves to the last child panel.
Enter Spacebar	Selects the child associated with the tab displaying focus.

Using Button controls to navigate an Accordion container

The simplest way for users to navigate the panels of an Accordion container is to click the navigator button for the desired panel. However, you can create additional navigation Button controls, such as Back and Next, to make it easier for users to navigate.

Navigation Button controls use the following properties of the Accordion container to move among the child panels:

Property	Description
`numChildren`	Contains the total number of child panels defined in an Accordion container.
`selectedIndex`	The index of the currently active child panel. Child panels are numbered from 0 to `numChildren` - 1. Setting the `selectedIndex` property changes the currently active panel.
`selectedChild`	The currently active child container if one or more child containers are defined. This property is `null` if no child containers are defined. Set this property to the identifier of the container that you want active. You can set this property only in an ActionScript statement, not in MXML.

For more information on these properties, see "ViewStack navigator container" on page 628.

You can use the following two Button controls, for example, in the second panel of an Accordion container, panel number 1, to move back to panel number 0 or ahead to panel number 2:

```
<?xml version="1.0"?>
<!-- containers\navigators\AccordionButtonNav.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Accordion id="accordion1" height="450">

        <mx:Form id="shippingAddress" label="1. Shipping Address">

            <mx:FormItem id="sfirstNameItem" label="First Name">
                <mx:TextInput id="sfirstName"/>
            </mx:FormItem>

        </mx:Form>

        <mx:Form id="billingAddress" label="2. Billing Address">
            <mx:Button id="backButton"
                label="Back"
                click="accordion1.selectedIndex=0;"/>
            <mx:Button id="nextButton"
                label="Next"
                click="accordion1.selectedIndex=2;"/>
        </mx:Form>

        <mx:Form id="creditCardInfo" label="3. Credit Card Information">
        </mx:Form>

    </mx:Accordion>
</mx:Application>
```

You can also use relative location with navigation buttons. The following Button controls move forward and backward through Accordion container panels based on the current panel number:

```
<mx:Button id="backButton" label="Back"
  click="accordion1.selectedIndex = accordion1.selectedIndex - 1;"/>

<mx:Button id="nextButton" label="Next"
  click="accordion1.selectedIndex = accordion1.selectedIndex + 1;"/>
```

For the Next Button control, you also can use the `selectedChild` property to move to the next panel based on the value of the `id` property of the panel's container, as the following code shows:

```
<mx:Button id="nextButton" label="Next"
  click="accordion1.selectedChild=creditCardInfo;"/>
```

The following Button control opens the last panel in the Accordion container:

```
<mx:Button id="lastButton" label="Last"
  click="accordion1.selectedIndex = accordion1.numChildren - 1;"/>
```

Handling child button events

The Accordion container can recognize an event when the user changes the current panel. The Accordion container broadcasts a `change` event when the user changes the child panel, either by clicking a button or pressing a key, such as the Page Down key.

NOTE

A `change` event is not dispatched when the child panel changes programmatically; for example, the `change` event is not dispatched when you use the buttons in "Using Button controls to navigate an Accordion container" to change panels. However, the `valueCommit` event is dispatched.

You can register an event handler for the `change` event using the `change` property of the `<mx:Accordion>` tag, or by registering the handler in ActionScript. The following example logs the change event to flashlog.txt each time the user changes panels:

```
<mx:Accordion id="accordion1" height="450" change="trace('change');">
```

Controlling the appearance of accordion buttons

The buttons on an Accordion container are rendered by the AccordionHeader class, which is a subclass of Button, and has the same style properties as the Button class.

To change the style of an Accordion button, call the Accordion class `getHeaderAt()` method to get a reference to a child container's button, and then call the button's `setStyle()` method to set the style. The following example uses this technique to set a different text color for each of the Accordion buttons:

```
<?xml version="1.0"?>
<!-- containers\navigators\AccordionStyling.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600"
    height="600"
    creationComplete="setButtonStyles();">

    <mx:Script>
        <![CDATA[
            public function setButtonStyles():void {
                comp.getHeaderAt(0).setStyle('color', 0xAA0000);
                comp.getHeaderAt(1).setStyle('color', 0x00AA00);
            }
        ]]>
    </mx:Script>

    <mx:Accordion id="comp">
        <mx:VBox label="first box">
            <mx:TextInput/>
            <mx:Button label="Button 1"/>
        </mx:VBox>
        <mx:VBox label="second box">
            <mx:TextInput/>
            <mx:Button label="Button 2"/>
        </mx:VBox>
    </mx:Accordion>
</mx:Application>
```

You can also control the appearance of the buttons by using the `headerStyleName` style property of the Accordion class. For more information, see Accordion in *Adobe Flex 2 Language Reference*.

PART 3

Customizing the User Interface

3

This part describes how to improve the user experience by adding functionality to your application, including information about localizing your application.

The following topics are included:

CHAPTER 17

Using Behaviors

17

Behaviors let you add animation and motion to your application in response to user or programmatic action. For example, you can use behaviors to cause a dialog box to bounce slightly when it receives focus, or to slowly fade in when it becomes visible.

This topic describes how to build behaviors into your applications using MXML and ActionScript, and describes the two parts of a behavior: triggers and effects.

Contents

About behaviors

A *behavior* is a combination of a trigger paired with an effect. A *trigger* is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. An *effect* is a visible or audible change to the target component that occurs over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component.

You can define multiple effects to play in response to a single trigger. For example, a pet store application might contain a Button control to select a pet category. When the user clicks the Button control, a window that contains breed names becomes visible. As the window becomes visible, it moves to the bottom-left corner of the screen, and it resizes from 100 by 100 pixels to 300 by 300 pixels (the effects).

By default, Adobe Flex components do not play an effect when a trigger occurs. To configure a component to use an effect, you associate an effect with a trigger.

NOTE

Triggers are not the same as events. For example, a Button control has both a `mouseDown` event and a `mouseDownEffect` trigger. The event initiates the corresponding effect trigger when a user clicks on a component. You use the `mouseDown` event property to specify the event listener that executes when the user clicks on the component. You use the `mouseDownEffect` trigger property to associate an effect with the trigger.

About applying behaviors

You create, configure, and apply effects to Flex components by using both MXML and ActionScript. With MXML, you associate effects with triggers as part of defining the basic behavior for your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define effect. -->
    <mx:WipeLeft id="myWL" duration="1000"/>

    <!-- Assign effect to targets. -->
    <mx:Button id="myButton" mouseDownEffect="{myWL}"/>
    <mx:Button id="myOtherButton" mouseDownEffect="{myWL}"/>

</mx:Application>
```

In this example, the effect is a WipeLeft effect with a duration of 1000 milliseconds (ms). That means it takes 1000 ms for the effect to play from start to finish.

You use data binding to assign the effect to the `mouseDownEffect` property of each Button control. The `mouseDownEffect` property is the effect trigger that specifies to play the effect when the user clicks the control using the mouse pointer. In the previous example, the effect makes the Button control appear as if it is being wiped onto the screen from right to left.

Using ActionScript, you can create, modify, or play an effect. With ActionScript, you can configure the effect to play in response to an effect trigger, or you can explicitly invoke it by calling the `play()` method of the effect's class. ActionScript gives you control of effects so that you can configure them as part of a user preferences setting, or modify them based on user actions. The following example creates the WipeLeft effect in ActionScript:

```
<?xml version="1.0"?>
<!-- behaviors\AsEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createEffect(event);" >

    <!-- Define effect. -->
    <mx:Script>
        <![CDATA[

            // Import the effect class.
            import mx.effects.*;

            // Define effect variable.
            private var myWL:WipeLeft;

            private function createEffect(eventObj:Event):void {
                // Create the WipeLeft effect object.
                myWL=new WipeLeft();

                // Set the effect duration.
                myWL.duration=1000;

                // Assign the effects to the targets.
                myButton.setStyle('mouseDownEffect', myWL);
                myOtherButton.setStyle('mouseDownEffect', myWL);
            }
        ]]>
    </mx:Script>

    <mx:Button id="myButton"  />
    <mx:Button id="myOtherButton" />

</mx:Application>
```

This example still uses an event to invoke the effect. To play an effect programmatically, you call the effect's `play()` method. For information on using ActionScript to configure and invoke effects, and for more information on using MXML, see "Applying behaviors" on page 659.

About factory and instance classes

Flex implements effects using an architecture in which each effect is represented by two classes: a factory class and an instance class.

Factory class The factory class creates an object of the instance class to perform the effect on the target. You create a factory class instance in your application, and configure it with the necessary properties to control the effect, such as the zoom size or effect duration. You then assign the factory class instance to an effect trigger of the target component, as the following example shows:

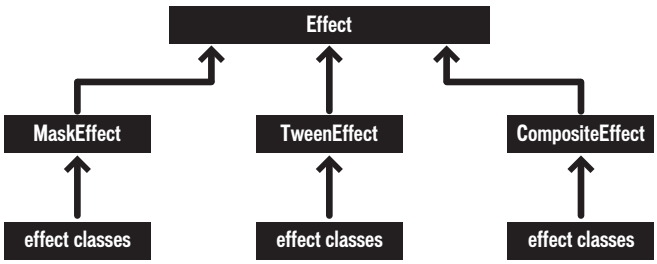
```
<!-- Define factory class. -->
<mx:WipeDown id="myWD" duration="1000"/>
<!-- Assign factory class to effect targets. -->
<mx:Button id="myButton" mouseDownEffect="{myWD}"/>
<mx:Button id="myOtherButton" mouseDownEffect="{myWD}"/>
```

By convention, the name of a factory class is the name of the effect, such as Zoom or Fade.

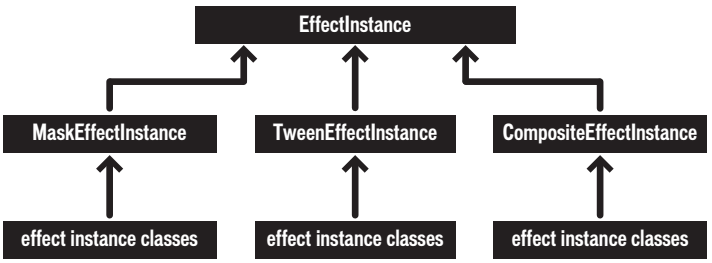
Instance class The instance class implements the effect logic. When an effect trigger occurs, or when you call the `play()` method to invoke an effect, the factory class creates an object of the instance class to perform the effect on the target. When the effect ends, Flex destroys the instance object. If the effect has multiple target components, the factory class creates multiple instance objects, one per target.

By convention, the name of an instance class is the name of the effect with the suffix *Instance*, such as ZoomInstance or FadeInstance.

The following diagram shows the class hierarchy of the Flex effect classes:



Factory classes



Instance classes

As you can see in this diagram, there is a corresponding instance class for each factory class.

When you use effects, you perform the following steps:

1. Create a factory class object.
2. Configure the factory class object.

When Flex plays an effect, Flex performs the following actions:

1. Creates one object of the instance class for each target component of the effect.
2. Copies configuration information from the factory object to the instance object.
3. Plays the effect on the target using the instance object.
4. Deletes the instance object when the effect completes.

Any changes that you make to the factory object are not propagated to a currently playing instance object. However, the next time the effect plays, the instance object uses your new settings.

When you use effects in your application, you are concerned only with the factory class; the instance class is an implementation detail. However, if you want to create custom effects classes, you must implement a factory and an instance class. For more information, see Chapter 15, "Creating Effects," in *Creating and Extending Flex 2 Components*.

Available effects

The following table lists the effects that Flex supports:

Effect	Description
AnimateProperty	Animates a numeric property of a component, such as `height`, `width`, `scaleX`, or `scaleY`. You specify the property name, start value, and end value of the property to animate. The effect first sets the property to the start value, and then updates the property value over the duration of the effect until it reaches the end value. For example, if you want to change the width of a Button control, you can specify width as the property to animate, and starting and ending width values to the effect.
Blur	Applies a blur visual effect to a component. A Blur effect softens the details of an image. You can produce blurs that range from a softly unfocused look to a Gaussian blur, a hazy appearance like viewing an image through semi-opaque glass. The Blur effect uses the Flash BlurFilter class as part of its implementation. For more information, see flash.filters.BlurFilter in *Adobe Flex 2 Language Reference*. If you apply a Blur effect to a component, you cannot apply a BlurFilter or a second Blur effect to the component.

Effect	Description
Dissolve	Modifies the `alpha` property of an overlay to gradually have to target component appear or disappear. The Dissolve effect has the following properties: • `alphaFrom` Specifies the initial alpha level (0.0=transparent, 1.0=fully opaque). If omitted, Flex uses the component's current alpha level. • `alphaTo` Specifies the final alpha level. • `color` Value that represents the color of the overlay rectangle that the effect displays over the target object. The default value is the color specified by the target component's `backgroundColor` style property, or 0xFFFFFF, if `backgroundColor` is not set. If the target object is a container, only the children of the container dissolve. The container borders do not dissolve. ***Note:*** To use the Dissolve effect with the `creationCompleteEffect` trigger of a DataGrid control, you must define the data provider of the control inline using a child tag of the DataGrid control, or using data binding. This issue is a result of the data provider not being set until the `creationComplete` event is dispatched. Therefore, when the effect starts playing, Flex has not completed the sizing of the DataGrid control. ***Note:*** To use the Dissolve effect with text, you must use an embedded font, not a device font. For more information, see Chapter 18, "Using Styles and Themes," on page 697.
Fade	Animate the component from transparent to opaque, or from opaque to transparent. The Fade effect has the following properties: • `alphaFrom` Specifies the initial alpha level (0.0=transparent, 1.0=fully opaque). If omitted, Flex uses the component's current alpha level. • `alphaTo` Specifies the final alpha level. If you specify the `Fade` effect for the `showEffect` or `hideEffect` trigger, and if you omit the `alphaFrom` and `alphaTo` properties, the effect automatically transitions from 0.0 to the targets' current `alpha` value for a show trigger, and from the targets' current `alpha` value to 0.0 for a hide trigger. ***Note:*** To use the Fade effect with text, you must use an embedded font, not a device font. For more information, see Chapter 18, "Using Styles and Themes," on page 697.
Glow	Applies a glow visual effect to a component. The Glow effect uses the Flash GlowFilter class as part of its implementation. For more information, see the flash.filters.GlowFilter class in the *Adobe Flex 2 Language Reference*. If you apply a Glow effect to a component, you cannot apply a GlowFilter or a second Glow effect to the component.

Effect	Description
Iris	Animates the effect target by expanding or contracting a rectangular mask centered on the target. The effect can either grow the mask from the center of the target to expose the target, or contract it toward the target center to obscure the component. For more information, see "Using a mask effect" on page 682.
Move	Changes the position of a component over a specified time interval. You typically apply this effect to a target in a container that uses absolute positioning, such as Canvas, or one with "`layout=absolute`", such as Application or Panel. If you apply it to a target in a container that performs automatic layout, such as a VBox or Grid container, the move occurs, but the next time the container updates its layout, it moves the target back to its original position. You can set the container's `autoLayout` property to `false` to disable the move back, but that disables layout for all controls in the container. The Move effect has the following properties: • `xFrom` and `yFrom` Specifies the initial position of the component. If omitted, Flex uses the current position. • `xTo` and `yTo` Specifies the destination position. • `xBy` and `yBy` Specifies the number of pixels to move the component in the x and y directions. Values can be positive or negative. For the `xFrom`, `xTo`, and `xBy` properties, you can specify any two of the three values; Flex calculates the third. If you specify all three properties, Flex ignores the `xBy` property. The same is true for the `yFrom`, `yTo`, and `yBy` properties. If you specify a Move effect for a trigger, and if you do not set the six From, To, and By properties, Flex sets them to create a smooth transition between the object's old position and its new position. When the Move effect runs, the layout around the component that is moving does not readjust. Setting a container's `autoLayout` property to `true` has no effect on this behavior.
Pause	Does nothing for a specified period of time. This effect is useful when you need to composite effects. For more information, see "Creating composite effects" on page 675.

Effect	Description
Resize	Changes the width and height of a component over a specified time interval. The Resize effect has the following properties: • `widthFrom` and `heightFrom` Specifies the initial width and height. If omitted, Flex uses the current size. • `widthTo` and `heightTo` Specifies the final width and height. • `widthBy` and `heightBy` Specifies the number of pixels to modify the size as either a positive or negative number relative to the initial width and height. For `widthFrom`, `widthTo`, and `widthBy`, you can specify any two of the three values, and Flex calculates the third. If you specify all three, Flex ignores `widthBy`. The same is true for `heightFrom`, `heightTo`, and `heightBy`. If you specify a Resize effect for a resize trigger, and if you do not set the six From, To, and By properties, Flex sets them to create a smooth transition between the object's old size and its new size. When you apply a Resize effect, the layout manager resizes neighboring components based on the size changes to the target component. To run the effect without resizing other components, place the target component in a Canvas container. When you use the Resize effect with Panel containers, you can hide Panel children to improve performance. For more information, see "Improving performance when resizing Panel containers" on page 694.
Rotate	Rotates a component around a specified point. You can specify the coordinates of the center of the rotation, and the starting and ending angles of rotation. You can specify positive or negative values for the angles. The Resize effect has the following properties: • `angleFrom` and `angleTo` Specifies the initial rotation and final rotation angles. • `originX` and `originY` Specifies coordinate of the center point of the rotation. ***Note:*** To use the Rotate effect with text, you must use an embedded font, not a device font. For more information, see Chapter 18, "Using Styles and Themes," on page 697.
SoundEffect	Plays an MP3 audio file. For example, you could play a sound when a user clicks a Button control. This effect lets you repeat the sound, select the source file, and control the volume and pan. You specify the MP3 file using the `source` property. If you have already embedded the MP3 file, using the `Embed` keyword, then you can pass the Class object of the MP3 file to the source property. Otherwise, specify the full URL to the MP3 file. For more information, see "Using a sound effect" on page 680.

Effect	Description
WipeLeft WipeRight WipeUp WipeDown	Defines a bar Wipe effect. The before or after state of the component must be invisible. These effects have the following property: • `showTarget` If `true`, causes the component to appear. If `false`, causes the component to disappear. The default value is `true`. If you specify a Wipe effect for a `showEffect` or `hideEffect` trigger, by default, Flex sets the `showTarget` property to `true` if the component is invisible, and `false` if the component is visible. For more information, see "Using a mask effect" on page 682.
Zoom	Zooms a component in or out from its center point. The Zoom effect has the following properties: • `zoomHeightFrom`, `zoomWidthFrom` Specifies a number that represents the scale at which to start the zoom. The default value is 0.0. • `zoomHeightTo`, `zoomWidthTo` Specifies a number to zoom to. The default value is 1.00. Specify a value of 2.0 to double the size of the target. • `originX`, `originY` The x-position and y-position of the origin, or *`registration`* point, of the zoom. The default value is the coordinates of the center of the effect target. ***Note:*** When you apply a Zoom effect to text rendered using a system font, Flex scales the text between whole point sizes. Although you do not have to use embedded fonts when you apply a Zoom effect to text, the Zoom will appear smoother when you apply it to embedded fonts. For more information, see Chapter 18, "Using Styles and Themes," on page 697.

Available triggers

You use a trigger name to assign an effect to a target component. You can reference a trigger name as a property of an MXML tag, in the `<mx:Style>` tag, or in an ActionScript `setStyle()` and `getStyle()` function. Trigger names use the following naming convention:

`triggerEvent``Effect`

where *`triggerEvent`* is the event that invokes the effect. For example, the `focusIn` event occurs when a component gains focus; you use the `focusInEffect` trigger property to specify the effect to invoke for the `focusIn` event. The `focusOut` event occurs when a component loses focus; the corresponding trigger property is `focusOutEffect`.

The following table lists the effect name that corresponds to each trigger:

Trigger name	Triggering event
`addedEffect`	Component is added as a child to a container.
`creationCompleteEffect`	Component is created.

Trigger name	Triggering event
`focusInEffect`	Component gains keyboard focus.
`focusOutEffect`	Component loses keyboard focus.
`hideEffect`	Component becomes invisible by changing the `visible` property of the component from `true` to `false`.
`mouseDownEffect`	User presses the mouse button while the mouse pointer is over the component.
`mouseUpEffect`	User releases the mouse button.
`moveEffect`	Component is moved.
`removedEffect`	Component is removed from a container.
`resizeEffect`	Component is resized.
`rollOutEffect`	User rolls the mouse pointer off the component.
`rollOverEffect`	User rolls the mouse pointer over the component.
`showEffect`	Component becomes visible by changing the `visible` property of the component from `false` to `true`.

Applying behaviors

This section describes how to apply behaviors using both MXML and ActionScript.

Applying behaviors in MXML

In MXML, you use a trigger name as a property of a component's MXML tag to configure an effect for the component. For example, to configure a Button control to use the WipeLeft effect when the user clicks the control, you use the `mouseDownEffect` trigger property in an `<mx:Button>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define effect. -->
    <mx:WipeLeft id="myWL" duration="1000"/>

    <!-- Assign effect to targets. -->
    <mx:Button id="myButton" mouseDownEffect="{myWL}"/>
    <mx:Button id="myOtherButton" mouseDownEffect="{myWL}"/>

</mx:Application>
```

In the next example, you create two Resize effects for a Button control. One Resize effect expands the size of the button by 10 pixels when the user clicks down on the button, and the second resizes it back to its original size when the user releases the mouse button. The duration of each effect is 200 ms.

```
<?xml version="1.0"?>
<!-- behaviors\ButtonResize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Resize id="myResizeUp"
        widthBy="50" heightBy="50"
        duration="200"/>
    <mx:Resize id="myResizeDown"
        widthBy="-50" heightBy="-50"
        duration="200"/>

    <mx:Button id="myButton"
        mouseDownEffect="{myResizeUp}"
        mouseUpEffect="{myResizeDown}"/>

</mx:Application>
```

Applying behaviors in MXML using data binding

You can use data binding in MXML to set properties of an effect. For example, the following example lets the user set the `zoomHeightTo` and `zoomWidthTo` properties of the Zoom effect using a TextInput control. The `zoomHeightTo` and `zoomWidthTo` properties specify a number that represents the scale at which to complete the zoom, as a value between 0.0 and 1.0. The default value is 1.0, which is the object's normal size.

```
<?xml version="1.0"?>
<!-- behaviors\DatabindingEffects.mxml  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Zoom id="mZoom"
        zoomHeightTo="{Number(zoomHeightInput.text)}"
        zoomWidthTo="{Number(zoomWidthInput.text)}"/>

    <mx:TextInput id="zoomHeightInput" text="1.0"/>
    <mx:TextInput id="zoomWidthInput" text="1.0"/>

    <mx:Button rollOverEffect="{mZoom}"/>

</mx:Application>
```

By default, the `text` property of the TextInput controls is set to 1.0. The user can edit the TextInput controls to specify a different zoom value.

Applying behaviors in MXML using styles

All MXML properties corresponding to effect triggers are implemented as CSS styles. Therefore, you can also apply an effect using the `<mx:Style>` tag. For example, to set the `mouseDownEffect` property for all TextArea controls in an application, you can use a CSS type selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\MxmlTypeSel.mxml-->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        TextArea { mouseDownEffect: slowWipe; }
    </mx:Style>

    <mx:WipeLeft id="slowWipe" duration="5000"/>
    <mx:TextArea id="myTA"/>

    <mx:TextArea id="myTA2" mouseDownEffect="none"/>

</mx:Application>
```

Setting the `mouseDownEffect` property in a component tag overrides any settings that you make in an `<mx:Style>` tag. If you want to remove the associated effect defined in a type selector, you can explicitly set the value of any trigger to `none`, as the following example shows:

```
<mx:TextArea id="myTA" mouseDownEffect="none"/>
```

You can also use a class selector to apply effects, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWLClassSel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Style>
        <!-- Define a class selector for a TextArea control -->
        .textAreaStyle{ mouseDownEffect: WipeLeft; }
    </mx:Style>

    <mx:WipeLeft id="slowWipe" duration="5000"/>
    <mx:TextArea id="myTA" styleName="textAreaStyle" />

</mx:Application>
```

Using setStyle() and getStyle() with behaviors defined in MXML

Trigger properties are implemented as styles; therefore, you can use the `setStyle()` and `getStyle()` methods to manipulate triggers and their associated effects. The `setStyle()` method has the following signature:

`setStyle("trigger_name", effect)`

trigger_name String indicating the name of the trigger property; for example, `mouseDownEffect` or `focusInEffect`.

effect The effect associated with the trigger. The data type of `effect` is an Effect object, or an object of a subclass of the Effect class.

The `getStyle()` method has the following signature:

`getStyle("trigger_name"):return_type`

trigger_name String indicating the name of the trigger property.

return_type An Effect object, or an object of a subclass of the Effect class.

The following scenarios show how to use `getStyle()` with behaviors defined in MXML:

When you use MXML tag properties or the `<mx:Style>` tag to apply effects to a target, `getStyle()` returns an Effect object. The type of the object depends on the type of the effect that you specified, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ButtonWLGetStyleMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <!-- Set the behavior in MXML. -->
    <mx:WipeLeft id="slowWipe" duration="5000"/>
    <mx:Button id="myB" mouseDownEffect="{slowWipe}" />

    <!-- Call getStyle() to return an object of type WipeLeft. -->
    <mx:Button label="get style"
        click="myTA.text=String(myB.getStyle('mouseDownEffect').duration);"
        />
    <mx:TextArea id="myTA" />

</mx:Application>
```

For more information on working with styles, see Chapter 18, “Using Styles and Themes,” on page 697.

Applying behaviors in ActionScript

You can declare and play effects in ActionScript, often as part of an event listener. To invoke the effect, you call the effect's `play()` method.

This technique is useful for using one control to invoke an effect on another control. The following example uses the event listener of a Button control's `click` event to invoke a Resize effect on an TextArea control:

```
<?xml version="1.0"?>
<!-- behaviors\ASplay.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createEffect(event);" >

    <mx:Script>
        <![CDATA[

            // Import effect class.
            import mx.effects.Resize;

            // Create a Resize effect
            private var resizeLarge:Resize = new Resize();

            private function createEffect(eventObj:Event):void {
                // Set the TextArea as the effect target.
                resizeLarge.target=myTA;

                // Set resized width and height, and effect duration.
                resizeLarge.widthTo=150;
                resizeLarge.heightTo=150;
                resizeLarge.duration=750;
            }
        ]]>
    </mx:Script>

    <mx:VBox borderStyle="solid">
        <mx:Button label="Start"
            click="resizeLarge.end();resizeLarge.play();"/>
        <mx:TextArea id="myTA" text="Here is some text."/>
    </mx:VBox>
</mx:Application>
```

In this example, use the application's `creationComplete` event to configure the effect, and then invoke it by calling the `play()` method in response to a user clicking the Button control. Because you invoke the effect programmatically, you do not need an event trigger to invoke it.

Notice that this example first calls the `Effect.end()` method before it calls the `play()` method. You call the `end()` method before you call the `play()` method to ensure that any previous instance of the effect has ended before you start a new one.

NOTE

Calling the `end()` method is only necessary when you invoke an effect using the `play()` method. If you invoke an effect using an effect trigger, such as the `mouseDownEffect` trigger, Flex makes sure to end the effect before starting a new one.

This example also defines the effect using ActionScript. You are not required to define the effect in ActionScript. You could rewrite this code using MXML to define the effect, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\MxmlPlay.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Resize id="resizeLarge" target="{myTA}"
        widthTo="150" heightTo="150" duration="750" />

    <mx:Canvas height="300" width="400" borderStyle="solid">
        <mx:Button label="Start" x="50" y="50"
            click="resizeLarge.end();resizeLarge.play();" />
        <mx:TextArea x="100" y="100" id="myTA" text="Here is some text."/>
    </mx:Canvas>
</mx:Application>
```

Optionally, you can pass an Array of targets to the `play()` method to invoke the effect on all components specified in the Array, as the following example shows:

```
resizeLarge.play([comp1, comp2, comp3]);
```

This example invokes the Zoom effect on three components. Notice that the `end()` method does not take an effect target as an argument but an effect instance. Therefore, you end this effect by calling the `end()` method on the effect itself, as the following example shows:

```
resizeLarge.end();
```

When you call the `play()` method, you essentially replace the effect trigger with the method call. You use the `Effect.target` or `Effect.targets` properties to specify the target components of an effect when you invoke them using the `play()` method. This example uses the `target` property of the effect to specify the single target component. If you want to play the effect on multiple components, you can use the `Effects.targets` property to specify an array of target components. For more information, see "Applying behaviors using the Effect.target and Effect.targets properties" on page 672.

Rather than using the `target` property to specify the target component of the effect, you can also pass the target component to the constructor, as the following example shows:

```
// Create a Resize effect.
var resizeLarge = new mx.effects.Resize(myTA);
```

Playing an effect backward

You can pass an optional argument to the `play()` method to play the effect backward, as the following example shows:

```
resizeLarge.play([comp1, comp2, comp3], true);
```

In this example, you specify `true` as the second argument to play the effect backward. The default value is `false`.

You can also use the `Effect.pause()` method to pause an effect, the `Effect.resume()` method to resume a paused effect, and the `Effect.reverse()` method to play an effect backward.

Ending an effect

You can use the `end()` method to terminate an effect at any time, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ASend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createEffect(event);" >

    <mx:Script>
        <![CDATA[

            // Import effect class.
            import mx.effects.Resize;

            // Create a Resize effect
            private var resizeLarge:Resize = new Resize();

            private function createEffect(eventObj:Event):void {
                // Set the TextArea as the effect target.
                resizeLarge.target=myTA;

                // Set resized width and height, and effect duration.
                resizeLarge.widthTo=150;
                resizeLarge.heightTo=150;
                // Set a long duration.
                resizeLarge.duration=10000;
            }
        ]]>
    </mx:Script>

    <mx:Canvas height="300" width="500" borderStyle="solid">
        <mx:Button label="Start"
            x="10" y="10"
            click="resizeLarge.end();resizeLarge.play();"/>
        <mx:Button label="End"
            x="10" y="50"
            click="resizeLarge.end();"/>
        <mx:TextArea id="myTA"
            x="100" y="100"
            text="Here is some text."/>
    </mx:Canvas>
</mx:Application>
```

In this example, you set the `duration` property of the Resize effect to 10 seconds, and add a new Button control that uses the `end()` method to terminate the effect when the user clicks the button.

When you call the `end()` method, the effect jumps to its end state and then terminates. In the case of the Resize effect, the effect sets the final size of the expanded TextArea control before it terminates, just as if you had let the effect finish playing. If the effect was a Move effect, the target component moves to its final position before terminating.

If you defined a listener for the `effectEnd` event, that listener gets invoked by the `end()` method, just as if you had let the effect finish playing. For more information on working with effect events, see "Handling effect events" on page 685.

Creating a reusable effect

The next example creates a reusable function that takes three arguments corresponding to the target of a Move effect and the coordinates of the move. The function then creates the Move effect and plays it on the target:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- behaviors\PlayEffectPassParams.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;
            import mx.effects.Effect;
            import mx.effects.Move;

            private var myMove:Move = new Move();

            // Click event listener that passes the target component
            // and the coordinates of the center of the parent container
            // to the function that creates the effect.
            private function playMove(target:Object,
                    newX:Number, newY:Number):void {
                myMove.end();
                myMove.target=target;
                myMove.duration = 1000;
                myMove.xTo = newX - target.width/2;
                myMove.yTo = newY - target.height/2;
                myMove.play();
            }

            // Create the Move effect and play it on the target
            // component passed to the function.
            private function handleClick(eventObj:Event):void {
                var targetComponent:Object = eventObj.currentTarget;
                var parentCont:Object = targetComponent.parent;
                playMove(eventObj.target, parentCont.width/2,
                    parentCont.height/2);
            }
        ]]>
    </mx:Script>

    <mx:Canvas width="200" height="200">
        <mx:Button id="myButton"
            label="Center me"
            click="handleClick(event);"/>
    </mx:Canvas>
</mx:Application>
```

Applying behaviors in ActionScript using styles

Because Flex implements the properties corresponding to effect triggers as styles, you can use style sheets and the `setStyle()` and `getStyle()` methods to apply effects. Therefore, you can create an effect in ActionScript, and then use the `setStyle()` method to associate it with a trigger, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ASStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createEffect(event);">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;
            import mx.effects.Zoom;

            private function createEffect(eventObj:Event):void {
                // Define a new Zoom effect.
                var zEffect:Zoom = new Zoom();
                zEffect.duration = 2000;
                zEffect.zoomHeightTo = 1.50;
                zEffect.zoomWidthTo = 1.50;

                // Apply the Zoom effect to the Button control.
                myB.setStyle("mouseDownEffect", zEffect);
            }
        ]]>
    </mx:Script>

    <mx:Button id="myB"/>
</mx:Application>
```

You can also define the effect in MXML, then use ActionScript to apply it, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\ASStylesMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initializeEffect(event);">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private function initializeEffect(eventObj:Event):void {
                myB.setStyle("mouseDownEffect", myWL);
            }
        ]]>
    </mx:Script>

    <mx:WipeLeft id="myWL" duration="1000" />

    <mx:Button id="myB"/>
</mx:Application>
```

The code in the following example alternates the WipeRight and WipeLeft effects for the `mouseDownEffect` style of a Button control:

```
<?xml version="1.0"?>
<!-- behaviors\ASStyleGetStyleMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            private function changeEffect():void {
                if (myButton.getStyle("mouseUpEffect") == myWR) {
                    myButton.setStyle("mouseUpEffect", myWL);
                }
                else if (myButton.getStyle("mouseUpEffect") == myWL) {
                    myButton.setStyle("mouseUpEffect", myWR);
                }
            }
        ]]>
    </mx:Script>

    <mx:WipeRight id="myWR" duration="1000"/>
    <mx:WipeLeft id="myWL" duration="1000"/>

    <mx:Button id="myButton"
        label="My Button"
        click="changeEffect();"
        mouseUpEffect="{myWL}"/>
</mx:Application>
```

Applying behaviors using the Effect.target and Effect.targets properties

You can use the `Effect.target` or `Effect.targets` properties to specify the effect targets, typically when you use the `play()` method to invoke the effect, rather than a trigger. You use the `Effect.target` property in MXML to specify a single target, and the `Effect.targets` property to specify an array of targets, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\TargetProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Zoom id="myZoom"
        zoomHeightFrom="0.10" zoomWidthFrom="0.10"
        zoomHeightTo="1.00" zoomWidthTo="1.00"
        target="{myButton}"/>

    <mx:Button id="myButton"
        label="Zoom target"
        click="myZoom.end();myZoom.play();"/>
</mx:Application>
```

In this example, you use data binding to the `target` property to specify that the Button control is the target of the Zoom effect. However, you do not associate the effect with a trigger, so you must call the effect's `play()` method to invoke it.

In the next example, you apply a Zoom effect to multiple Button controls by using data binding with the effect's `targets` property:

```
<?xml version="1.0"?>
<!-- behaviors\TargetProp3Buttons.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Zoom id="myZoom"
        zoomHeightFrom="0.10" zoomWidthFrom="0.10"
        zoomHeightTo="1.00" zoomWidthTo="1.00"
        targets="{[myButton1, myButton2, myButton3]}"/>

    <mx:Button id="myButton1"/>
    <mx:Button id="myButton2"/>
    <mx:Button id="myButton3"/>

    <mx:Button label="Zoom targets" id="myButton4"
        click="myZoom.end();myZoom.play();"/>
</mx:Application>
```

You do not define a trigger to invoke the effect, so you must call the effect's `play()` method to invoke it. Because you specified three targets to the effect, the `play()` method invokes the effect on all three Button controls.

You can also set the `target` and `targets` properties in ActionScript, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\TargetPropAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createEffect(event);">

    <mx:Script>
        <![CDATA[

            import mx.effects.Fade;
            import flash.events.Event;

            private var myFade:Fade = new Fade();

            private function createEffect(eventObj:Event):void {
                myFade.duration=5000;

                // Pass Array of targets to play().
                myFade.play([myPanel1, myPanel2]);

                // Alternatively, set targets to an array of components.
                // myFade.targets = [myPanel1, myPanel2];
                // myFade.play();
            }

            private function playZoom(eventObj:Event):void {
                myZoom.end();

                // Alternatively, pass Array of targets to play().
                // myZoom.play([myTA]);

                // Set target to a single component.
                myZoom.target = myTA;
                myZoom.play();
            }
    ]]>
    </mx:Script>

    <mx:Zoom id="myZoom" duration="2000"
        zoomHeightFrom="0.10" zoomWidthFrom="0.10"
        zoomHeightTo="1.00" zoomWidthTo="1.00"/>

    <mx:Panel id="myPanel1" >
        <mx:TextArea id="myTA"/>
    </mx:Panel>

    <mx:Panel id="myPanel2" >
        <mx:Button id="myButton" click="playZoom(event);"/>
```

```
    </mx:Panel>
</mx:Application>
```

In this example, you use the `targets` property of the Fade effect to specify the two Panel containers as the effect target, and the `target` property of the Zoom effect to specify the Button control as the single target.

If you use the `targets` property to define multiple event targets, and you want to later use the `end()` method to terminate the effect, you must save the return value of the `play()` method and pass it as an argument to the `end()` method, as the following example shows:

```
var myFadeArray:Array = myFade.play();
```

Then you can pass the array to the `end()` method to end the effect on all targets, as the following example shows:

```
myFade.end(myFadeArray);
```

Working with effects

This section describes how to handle effect events, how to customize effects, and advanced techniques for using effects.

Setting effect durations

All effects take the `duration` property that you can use to specify the time, in milliseconds, during which the effect occurs. The following example creates two new versions of the WipeLeft effect. The SlowWipe effect uses a two-second duration; the ReallySlowWipe effect uses an eight-second duration:

```
<?xml version="1.0"?>
<!-- behaviors\WipeDuration.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:WipeLeft id="SlowWipe" duration="2000"/>
    <mx:WipeLeft id="ReallySlowWipe" duration="8000"/>

    <mx:Button label="Slow Wipe"
        mouseDownEffect="{SlowWipe}"/>
    <mx:Button label="Really Slow Wipe"
        mouseDownEffect="{ReallySlowWipe}"/>
</mx:Application>
```

Using embedded fonts with effects

The Dissolve, Fade, and Rotate effects only work with text rendered using an embedded font. If you apply these effects to a control that uses a system font, nothing happens to the text.

When you apply a Zoom effect to text rendered using a system font, Flex scales the text between whole point sizes. While you do not have to use embedded fonts when you apply a Zoom effect to text, the Zoom will appear smoother when you apply it to embedded fonts.

The following example uses two Label controls, one that uses an embedded font and one that does not. Therefore, when you apply the Fade effect to the Label control using the system font, nothing happens:

```
<?xml version="1.0"?>
<!-- behaviors\EmbedFont.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        @font-face {
            src:url("../assets/MyriadWebPro.ttf");
            font-family: myMyriadWebPro;
        }
    </mx:Style>

    <mx:Fade id="fadeOut" alphaFrom="1.0" alphaTo="0.5"/>
    <mx:Fade id="fadeIn" alphaFrom="0.5" alphaTo="1.0"/>

    <mx:VBox>
        <mx:Label fontFamily="myMyriadWebPro"
            mouseDownEffect="{fadeOut}" mouseUpEffect="{fadeIn}"
            text="This Label control uses the MyriadWebPro embedded font
 and the text will fade."/>

        <mx:Label
            mouseDownEffect="{fadeOut}" mouseUpEffect="{fadeIn}"
            text="This Label control uses the system font
and the text will not fade."/>
    </mx:VBox>
</mx:Application>
```

Creating composite effects

Flex supports two ways to combine, or *composite*, effects:

Parallel The effects play at the same time.

Sequence One effect must complete before the next effect starts.

To define a Parallel or Sequence effect, you use the <mx:Parallel> or <mx:Sequence> tag. The following example defines the Parallel effect ZoomRotateShow, which combines the Zoom and Rotate effects in parallel, and ZoomRotateHide, which combines the Zoom and Rotate effects in sequence:

```
<?xml version="1.0"?>
<!-- behaviors\CompositeEffects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"  >

    <mx:Parallel id="ZoomRotateShow">
            <mx:Zoom id="myZoomShow"
                zoomHeightFrom="0.0" zoomWidthFrom="0.0"
                zoomHeightTo="1.0" zoomWidthTo="1.0"/>
            <mx:Rotate id="myRotateShow" />
    </mx:Parallel>

    <mx:Sequence id="ZoomRotateHide">
            <mx:Rotate id="myRotateHide"/>
            <mx:Zoom id="myZoomHide"
                zoomHeightFrom="1.0" zoomWidthFrom="1.0"
                zoomHeightTo="0.0" zoomWidthTo="0.0"/>
    </mx:Sequence>

    <mx:VBox id="myBox" height="100" width="200">
        <mx:TextArea id="aTextArea"
            text="hello"
            hideEffect="{ZoomRotateHide}"
            showEffect="{ZoomRotateShow}"/>
    </mx:VBox>

    <mx:Button id="myButton1"
        label="Show!"
        click="aTextArea.visible=true;"/>
    <mx:Button id="myButton2"
        label="Hide!"
        click="aTextArea.visible=false;"/>
</mx:Application>
```

The event listenr for the `click` event for the Button control alternates making the VBox container visible and invisible. When the VBox container becomes invisible, it uses the ZoomRotateShow effect as its hide effect, and when it becomes invisible, it uses the ZoomRotateHide effect.

Notice that the VBox container sets the `autoLayout` property to `false`. This setting prevents Flex from updating the layout of the container while the effect is playing. For more information, see "Disabling container layout for effects" on page 692.

You can nest <mx:Parallel> and <mx:Sequence> tags inside each other. For example, two effects can run in parallel, followed by a third effect running in sequence.

In a Parallel or Sequence effect, the `duration` property sets the duration of each effect. For example, if the a Sequence effect has its `duration` property set to 3000, then each effect in the Sequence will take 3000 ms to play.

You can also create an event listener that combines effects into a composite effect, and then plays the composite effect, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\CompositeEffectsAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createEffect(event);">

    <mx:Script>
        <![CDATA[

            // Import effect classes and create effect variables.
            import mx.effects.*;
            public var myZoomShow:Zoom;
            public var myRotateShow:Rotate;
            public var ZRShow:Parallel;

            private function createEffect(eventObj:Event):void {
                // Create a Zoom effect.
                myZoomShow=new Zoom(aTextArea);
                myZoomShow.zoomHeightFrom=0.0;
                myZoomShow.zoomWidthFrom=0.0;
                myZoomShow.zoomHeightTo=1.0;
                myZoomShow.zoomWidthTo=1.0;

                // Initialize a Rotate effect.
                myRotateShow=new Rotate(aTextArea);

                // Initialize a Parallel effect.
                ZRShow=new Parallel();
                ZRShow.addChild(myZoomShow);
                ZRShow.addChild(myRotateShow);
            }
        ]]>
    </mx:Script>

    <mx:VBox id="myBox" height="100" width="200">
        <mx:TextArea id="aTextArea" text="hello" visible="false"/>
    </mx:VBox>

    <mx:Button id="myButton1"
        label="Show!"
        click="aTextArea.visible=true; ZRShow.end(); ZRShow.play();"/>
</mx:Application>
```

In this example, you use the `Parallel.addChild()` method to add each effect to the Parallel effect, and you then invoke the effect using the `Effect.play()` method.

Using the AnimateProperty effect

You use the AnimateProperty effect to animates a numeric property of a component. For example, you can use this effect to animate the `scaleX` property of a control, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\AnimateHScrollPos.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Sequence id="animateScaleXUpDown" >
        <mx:AnimateProperty
            property="scaleX"
            fromValue="1.0"
            toValue="1.5"/>
        <mx:AnimateProperty
            property="scaleX"
            fromValue="1.5"
            toValue="1.0"/>
    </mx:Sequence>

    <mx:Button label="Scale Button"
        mouseDownEffect="{animateScaleXUpDown}"/>
</mx:Application>
```

In this example, clicking on the Button control starts the Sequence effect, which is made up of two AnimateProperty effects. The first AnimateProperty effect scales the control to 150% of its width, and the second scrolls it back to its original width.

Delaying effect start

The `Effect.startDelay` property specifies a value, in milliseconds, that the effect will wait once it is triggered before it begins. You can specify an integer value greater than or equal to 0. If you have used the `Effect.repeatCount` property to specify the number of times to repeat the effect, the `startDelay` property is applied only to the first time the effect plays, but not to the repeated playing of the effect.

If you set the `startDelay` property for a Parallel effect, Flex inserts the delay between each effect of the parallel effect.

Repeating effects

All effects support the `Effect.repeatCount` and `Effect.repeatDelay` properties that let you configure whether effects repeat, where:

- `repeatCount` Specifies the number of times to play the effect. A value of 0 means to play the effect indefinitely until stopped by a call to the `end()` method. The default value is 1. For a repeated effect, the `duration` property specifies the duration of a single instance of the effect. Therefore, if an effect has a `duration` property set to 2000, and a `repeatCount` property set to 3, then the effect takes a total of 6000 ms (6 seconds) to play.
- `repeatDelay` Specifies the amount of time, in milliseconds, to pause before repeating the effect. The default value is 0.

For example, the following example repeats the Rotate effect until the user clicks a Button control:

```
<?xml version="1.0"?>
<!-- behaviors\repeatEff.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Rotate id="myRotate"
        repeatCount="0"/>
    <mx:Image
        source="@Embed(source='../assets/myImage.jpg')"
        mouseDownEffect="{myRotate}"/>

    <mx:Button id="myButton" label="Stop Rotation"
        click="myRotate.end();"/>

</mx:Application>
```

All effects dispatch an `effectEnd` event when the effect completes. If you repeat the effect, the effect dispatches the `effectEnd` event after the final repetition.

If the effect is a tween effect, such as a Fade or Move effect, the effect dispatches both the `tweenEnd` effect and the `endEffect` when the effect completes. If you configure the tween effect to repeat, the `tweenEnd` effect occurs at the end of every repetition of the effect, and the `endEffect` event occurs after the final repetition.

Using effects with the ViewStack and TabNavigator containers

The ViewStack and TabNavigator containers are each made up of a collection of child containers that let you select the currently visible child container. When you change the currently visible child container, you can use the `hideEffect` property of the container being hidden and the `showEffect` property of the newly visible child container to apply effects to the child containers.

The ViewStack or TabNavigator container waits for the `hideEffect` of the child container being hidden to complete before it reveals the new child container. You can interrupt a currently playing effect if you change the `selectedIndex` property of the ViewStack or TabNavigator container while an effect is playing.

For more information on the ViewStack and TabNavigator container, see Chapter 16, "Using Navigator Containers," on page 627.

Using a sound effect

You use the SoundEffect class to play a sound represented as an MP3 file. You specify the MP3 file using the `source` property. If you have already embedded the MP3 file, using the `Embed` keyword, you can pass the Class object of the MP3 file to the source property. Otherwise, specify the full URL to the MP3 file.

The following example shows both methods of specifying the MP3 file:

```
<?xml version="1.0"?>
<!-- behaviors\Sound.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Embed MP3 file.
            [Bindable]
            [Embed(source="../assets/sound1.mp3")]
            public var soundClass:Class;
        ]]>
    </mx:Script>

    <mx:SoundEffect id="soundEmbed"
        useDuration="false"
        loops="0"
        source="{soundClass}"/>

    <mx:Button id="myButton2"
        label="Sound Embed"
        mouseDownEffect="{soundEmbed}"/>
</mx:Application>
```

In this example, you embed the sound1.mp3 file in your application. That means the file is compiled into the SWF file.

The SoundEffect class has several properties that you can use to control the playback of the MP3 file, including `useDuration` and `loops`. The `useDuration` property specifies whether to use the `duration` property to control the play time of the MP3 file. If the `useDuration` property is `true`, the MP3 file will play for as long as the time specified by the `duration` property, which defaults to 500 ms. If you set `useDuration` to `false`, the MP3 file plays to completion.

The `loops` property specified the number of times to repeat the MP3 file, where a value of 0 means play the effect once, a value of 1 means play the effect twice, and so on. If you repeat the MP3 file, it still uses the setting of the `useDuration` property to determine the playback time.

The `duration` property takes precedence over the `loops` property. If the effect duration is not long enough to play the sound at least once, then the sound will not loop.

The SoundEffect class also defines the following events:

complete Dispatched when the sound file completes loading.

id3 Dispatched when ID3 data is available for an MP3 sound file.

ioError Dispatched when an error occurs during the loading of the sound file.

progress Dispatched periodically as the sound file loads. Within the event object, you can access the number of bytes currently loaded and the total number of bytes to load. The event is not guaranteed to be dispatched, which means that the `complete` event might be dispatched without any `progress` events being dispatched.

For more information, see the *Adobe Flex 2 Language Reference*.

Using a mask effect

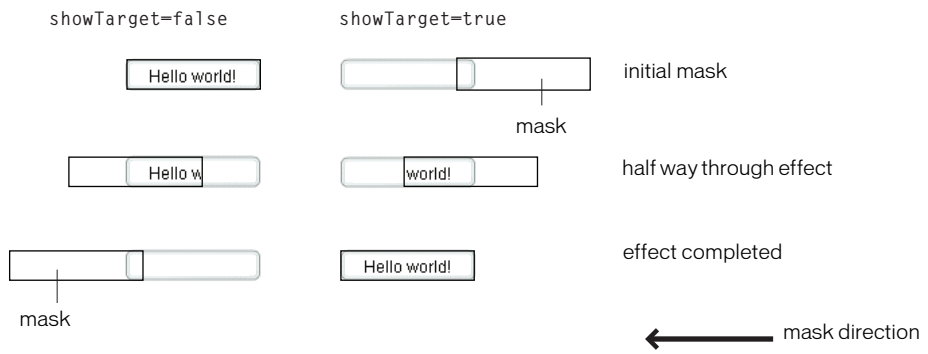
A mask effect is any effect that is a subclass of the MaskEffect class, which includes the wipe effects and the Iris effect. A mask effect uses an overlay, called a mask, to perform the effect. By default, for the wipe effects, the mask is a rectangle with the same size as the target component. For the Iris effect, the default mask is a rectangle centered on the component.

The before or after state of the target component of a mask effect must be invisible. That means a mask effect always makes a target component appear on the screen, or disappear from the screen.

To control the mask effect, you set the `MaskEffect.showTarget` property to correspond to the action of the component. If the target component is becoming visible, set `showTarget` to `true`. If the target is becoming invisible, set `showTarget` to `false`. The default value is `true`.

Often, you use these effects with the `showEffect` and `hideEffect` triggers The `showEffect` trigger occurs when a component becomes visible by changing its `visible` property from `false` to `true`. The `hideEffect` trigger occurs when the component becomes invisible by changing its `visible` property from `true` to `false`. When using a mask effect with the `showEffect` or `hideEffect` triggers, you can ignore the `showTarget` property; Flex sets it for you automatically.

As the mask effect executes, the effect either covers the target component or uncovers it, based on the setting of the `showTarget` property. The following diagram shows the action of the WipeLeft effect for the two different settings of the `showTarget` property:



You can use several properties of the MaskEffect class to control the location and size of the mask, including the following:

scaleXFrom, scaleYFrom, scaleXTo, and scaleX Specify the initial and final scale of the mask where a value of 1.0 corresponds to scaling the mask to the size of the target component, 2.0 scales the mask to twice the size of the component, 0.5 scales the mask to half the size of the component, and so on. To use any one of these properties, you must specify all four.

xFrom, yFrom, xTo, and yTo Specify the coordinates of the initial position and final position of the mask relative to the target component, where (0, 0) corresponds to the upper-left corner of the target. To use any one of these properties, you must specify all four.

The coordinates of the initial and final position of the mask depend on the type of effect and whether the `showTarget` property is `true` or `false`. For example, for the WipeLeft effect with a `showTarget` value of `false`, the coordinates of the initial mask position are (0, 0), corresponding to the upper-left corner of the target, and the coordinates of the final position are the upper-right corner of the target (`-width`, 0), where `width` is the width of the target.

For a `showTarget` value of `true` for the WipeLeft effect, the coordinates of the initial mask position are (`width`, 0), and the coordinates of the final position are (0, 0).

Creating a custom mask function

You can supply a custom mask function to a mask effect using the `createMaskFunction` property. A custom mask function lets you create a mask with a custom shape, color, or other attributes for your application requirements.

The custom mask function has the following signature:

```
public function funcName(targ:Object, bounds:Rectangle):Shape
  var myMask:Shape = new Shape();
  // Create mask.

  return myMask;
}
```

Your custom mask function takes an argument that corresponds to the target component of the effect, and a second argument that defines the dimensions of the target so that you can correctly size the mask. The function returns a single Shape object that defines the mask.

The following example uses a custom mask with a WipeLeft effect:

```
<?xml version="1.0"?>
<!-- behaviors\CustomMaskSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[

            // Import the effect class.
            import mx.effects.*;

            public function createLargeMask(targ:Object,
                    bounds:Rectangle):Shape {
                // Create the Shape object.
                var largeMask:Shape = new Shape();

                // Access the Graphics object of the
                // Shape object to draw the mask.
                largeMask.graphics.beginFill(0x00FFFF, 0.5);
                largeMask.graphics.drawRoundRect(0, 0, bounds.width + 10,
                    bounds.height - 10, 3);
                largeMask.graphics.endFill();

                // Return the mask.
                return largeMask;
            }
        ]]>
    </mx:Script>

    <mx:WipeLeft id="customWL"
        createMaskFunction="createLargeMask"
        showTarget="false"/>

    <mx:WipeLeft id="standardWL"
        showTarget="false"/>

    <mx:HBox borderStyle="solid"
        paddingLeft="10" paddingRight="10"
        paddingTop="10" paddingBottom="10">

        <mx:Button label="custom mask"
            mouseDownEffect="{customWL}"
            height="100" width="100"/>

    </mx:HBox>
</mx:Application>
```

Handling effect events

Every effect class supports the following events:

- `effectStart` Dispatched when the effect starts playing. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_START`.
- `effectEnd` Dispatched after the effect has stops playing, either when the effect finishes playing or when the effect has been interrupted by a call to the `end()` method. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_END`.

Flex dispatches one event for each target of an effect. Therefore, if you define a single target for an effect, Flex dispatches a single `effectStart` event, and a single `effectEnd` event. If you define three targets for an effect, Flex dispatches three `effectStart` events, and three `effectEnd` events.

The event object passed to the event listener for these event is of type EffectEvent. The EffectEvent class is a subclass of the Event class, and contains all of the properties inherited from Event, including `target`, and `type`, and defines a new property named `effectInstance`, where:

target Contains a reference to the Effect object that dispatched the event. This is the factory class of the effect.

type Either `EffectEvent.EFFECT_END` or `EffectEvent.EFFECT_START`, depending on the event.

effectInstance Contains a reference to the EffectInstance object. This is the object defined by the instance class for the effect. Flex creates one object of the instance class for each target of the effect. You access the target component of the effect using the `effectInstance.target` property.

The following example defines an event listener for the `endEffect` event:

```
<?xml version="1.0"?>
<!-- behaviors\EventEffects2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[

            import mx.effects.*;
            import mx.events.EffectEvent;
            import mx.core.UIComponent;

            private function endEffectListener(eventObj:EffectEvent):void {
                // Access the effect object.
                var effectObj:Effect = Effect(eventObj.target);

                // Access the target component of the effect.
                var effectTarget:UIComponent =
                    UIComponent(eventObj.effectInstance.target);

                myTA.text = effectTarget.id;
                myTA.text = myTA.text + " " + eventObj.type;
            }
        ]]>
    </mx:Script>

    <mx:Fade id="myFade" effectEnd="endEffectListener(event);"/>
    <mx:Button id="myButton" mouseUpEffect="{myFade}" />

    <mx:TextArea id="myTA" />
</mx:Application>
```

If the effect has multiple targets, Flex dispatches an `effectStart` event and `effectEnd` event once per target, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\EventEffects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[

            import mx.effects.*;
            import mx.events.EffectEvent;
            import mx.core.UIComponent;

            private function
endSlowFadeEffectListener(eventObj:EffectEvent):void
            {
                // Access the effect object.
                var effectObj:Effect = Effect(eventObj.target);

                // Access the target component of the effect.
                var effectTarget:UIComponent =
                    UIComponent(eventObj.effectInstance.target);

                myTA.text = myTA.text + effectTarget.id + '\n';
                myTA.text = myTA.text + " " + eventObj.type + '\n';
            }
        ]]>
    </mx:Script>

    <mx:Fade id="slowFade"
        duration="2000"
        effectEnd="endSlowFadeEffectListener(event);"/>

    <mx:Button id="myButton1" creationCompleteEffect="{slowFade}"/>
    <mx:Button id="myButton2" creationCompleteEffect="{slowFade}"/>
    <mx:Button id="myButton3" creationCompleteEffect="{slowFade}"/>
    <mx:Button id="myButton4" creationCompleteEffect="{slowFade}"/>

    <mx:TextArea id="myTA" height="200" width="100" />
</mx:Application>
```

Flex dispatches an `effectEnd` event once per target, therefore, the `endSlowFadeEffectListener()` event listener is invoked four times, once per Button control.

Handling tween effect events

Every effect class that is a subclass of the TweenEffect class, such as the Fade and Move effects, supports the following events:

- `tweenStart` Dispatched when the tween effect starts. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_START`. The `Effect.effectStart` event is dispatched before the `tweenStart` event.
- `tweenUpdate` Dispatched every time a TweenEffect class calculates a new value. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_UPDATE`.
- `tweenEnd` Dispatched when the tween effect ends. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_END`.

The event object passed to the event listener for these events is of type TweenEvent. The TweenEvent class is a subclass of the Event class, and contains all of the properties inherited from Event, including `target`, and `type`, and defines the following new property:

value Contains the tween value calculated by the effect. For example, for the Fade effect, the `value` property contains a single Number between the values of the `Fade.alphaFrom` and `Fade.alphaTo` properties. For the Move effect, the `value` property contains a two item Array, where the first value is the current `x` value of the effect target and the second value is the current `y` value of the effect target. For more information on the `value` property, see the instance class for each effect that is a subclass of the TweenEffect class.

Suspending background processing

To improve the performance of effects, you can disable background processing in your application for the duration of the effect by setting the `Effect.suspendBackgroundProcessing` property to `true`. The background processing that is blocked includes component measurement and layout, and responses to data services for the duration of the effect.

The default value of the `suspendBackgroundProcessing` property is `false`. You can set it to `true` in most cases. However, you should set it to `false` if either of the following conditions is true for your application:

- User input may arrive while the effect is playing, and the application must respond to the user input before the effect finishes playing.
- A response may arrive from the server while the effect is playing, and the application must process the response while the effect is still playing.

Using an easing function

You can change the speed of an animation by defining an easing function for an effect. With easing, you can create a more realistic rate of acceleration and deceleration. You can also use an easing function to create a bounce effect or control other types of motion.

Flex supplies predefined easing functions in the mx.effects.easing package. This package includes functions for the most common types of easing, including Bounce, Linear, and Sine easing. For more information on using these functions, see *Adobe Flex 2 Language Reference*.

NOTE

An easing function takes four arguments, following the function signature popularized by Robert Penner. For more information, see www.ericd.net/chapter7.pdf.

The following code shows the basic format of an easing function:

```
function myEasingFunction(t:Number, b:Number, c:Number, d:Number):Number {
  ...
}
```

You specify the following arguments of an easing function:

- `t` specifies time.
- `b` specifies the initial position of a component.
- `c` specifies the total change in position of the component.
- `d` specifies the duration of the effect, in milliseconds.

Using a Flex easing function

You specify an easing function to a component by passing a reference to the function to a component property. You only pass only the name of the easing function; Flex automatically sets the values for the arguments of the easing function.

All tween effects, meaning effect classes that are child classes of the TweenEffect class, support the `easingFunction` property, which lets you specify an easing function to the effect. Mask effects—those effect classes that are child classes of the MaskEffect class—also support easing functions. Other components support easing functions as well. For example, the Accordion and Tree components let you use the `openEasingFunction` style property to specify an easing function, and the ComboBox component supports a `closeEasingFunction` style property.

For example, you can specify the `mx.effects.easing.Bounce.easeOut()` method to the Accordion container using the `openEasingFunction` property, as the following code shows:

```
<?xml version="1.0"?>
<!-- behaviors\EasingFuncExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="550">

    <mx:Script>
        import mx.effects.easing.*;
    </mx:Script>

    <mx:Accordion
        openEasingFunction="{Bounce.easeOut}"
        openDuration="2000">
        <mx:VBox label="Pane 1" width="400" height="400"/>
        <mx:VBox label="Pane 2" width="400" height="400"/>
    </mx:Accordion>
</mx:Application>
```

Creating a custom easing function

In the following example, you create a custom easing function that creates a bounce motion when combined with the Flex Move effect:

```
<?xml version="1.0"?>
<!-- behaviors\Easing.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
            private function myEasingFunction(t:Number, b:Number,
                    c:Number, d:Number):Number {
              if ((t /= d) < (1 / 2.75)) {
                  return c * (7.5625 * t * t) + b;
              }
              else if (t < (2 / 2.75)) {
                  return c * (7.5625 * (t-=(1.5/2.75)) * t + .75) + b;
              }
              else if (t < (2.5 / 2.75)) {
                  return c * (7.5625 * (t-=(2.25/2.75)) * t + .9375) + b;
              }
              else {
                 return c * (7.5625 * (t-=(2.625/2.75)) * t + .984375) + b;
              }
            };
        ]]>
    </mx:Script>

    <mx:Move id="moveLeftShow"
        xFrom="600" xTo="0" yTo="0"
        duration="3000"
        easingFunction="myEasingFunction"/>
    <mx:Move id="moveRightHide"
        xFrom="0" xTo="600"
        duration="3000"
        easingFunction="myEasingFunction"/>

    <mx:LinkBar dataProvider="myVS"/>
    <mx:ViewStack id="myVS" borderStyle="solid">
        <mx:Canvas id="Canvas0" label="Canvas0"
            creationCompleteEffect="{moveLeftShow}"
            showEffect="{moveLeftShow}"
            hideEffect="{moveRightHide}" >
                <mx:Box height="300" width="600" backgroundColor="#00FF00">
                  <mx:Label text="Screen 0" color="#FFFFFF" fontSize="40"/>
                </mx:Box>
        </mx:Canvas>
        <mx:Canvas id="Canvas1" label="Canvas1"
            showEffect="{moveLeftShow}" hideEffect="{moveRightHide}" >
```

```
                <mx:Box height="300" width="600" backgroundColor="#0033CC">
                  <mx:Label text="Screen 1" color="#FFFFFF" fontSize="40"/>
                </mx:Box>
            </mx:Canvas>
    </mx:ViewStack>
</mx:Application>
```

In this example, you use the custom effects in the `showEffect` and `hideEffect` properties of the children of a ViewStack container. When you click a label in the LinkBar control, the corresponding child of the ViewStack container slides in from the right, and bounces to a stop against the left margin of the ViewStack container, while the previously visible child of the ViewStack container slides off to the right.

The custom effect for the `showEffect` property is only triggered when the child's visibility changes from `false` to `true`. Therefore, the first child of the ViewStack container also includes a `creationCompleteEffect` property. This is necessary to trigger the effect when Flex first creates the component. If you omit the `creationCompleteEffect` property, you do not see the `moveLeftShow` effect when the application starts.

Disabling container layout for effects

By default, Flex updates the layout of a container's children when a new child is added to it, when a child is removed from it, when a child is resized, and when a child is moved. Because the Move effect modifies a child's position, and the Zoom effect modifies a child's size and position, they both cause the container to update its layout.

However, when the container updates its layout, it can actually reverse the results of the effect. For example, you use the Move effect to reposition a container child. At some time later, you change the size of another container child, which forces the container to update its layout. This layout update can cause the child that moved to be returned to its original position.

To prevent Flex from performing layout updates, you can set the `autoLayout` property of a container to `false`. Its default value is `true`, which configures Flex so that it always updates layouts. You always set the `autoLayout` property on the parent container of the component that uses the effect. For example, if you want to control the layout of a child of a Grid container, you set the `autoLayout` property for the parent GridItem container of the child, not for the Grid container.

You set the `autoLayout` property to `false` when you use a Move effect in parallel with a Resize or Zoom effect. You must do this because the Resize or Zoom effect can cause an update to the container's layout, which can return the child to its original location.

When you use the Zoom effect on its own, you can set the `autoLayout` property to `false`, or you may leave it with its default value of `true`. For example, if you use a Zoom effect with the `autoLayout` property set to `true`, as the child grows or shrinks, Flex automatically updates the layout of the container to reposition its children based on the new size of the child. If you use a Zoom effect with the `autoLayout` property set to `false`, the child resizes around its center point, and the remaining children do not change position.

The HBox container in the following example uses the default vertical alignment of `top` and the default horizontal alignment of `left`. If you apply a Zoom effect to the image, the HBox container resizes to hold the image, and the image remains aligned with the upper-left corner of the container:

```
<mx:HBox>
  <mx:Image source="myImage.jpg"/>
</mx:HBox>
```

In the next example, the image is centered in the HBox container. If you apply a Zoom effect to the image, as it resizes, it remains centered in the HBox container.

```
<mx:HBox horizontalAlign="center" verticalAlign="middle">
  <mx:Image source="myImage.jpg"/>
</mx:HBox>
```

By default, the size of the HBox container is big enough to hold the image at it original size. If you disable layout updates, and use the Zoom effect to enlarge the image, or use the Move effect to reposition the image, the image might extend past the boundaries of the HBox container, as the following example shows:

```
<mx:HBox autoLayout="false">
  <mx:Image source="myImage.jpg"/>
</mx:HBox>
```

You set the `autoLayout` property to `false`, so the HBox container does not resize as the image resizes. If the image grows to a size so that it extends beyond the boundaries of the HBox container, the container adds scroll bars and clips the image at its boundaries.

To prevent the scroll bars from appearing, you can use the `height` and `width` properties to explicitly size the HBox container so that it is large enough to hold the modified image. Alternatively, you can set the `clipContent` property of the container to `false` so that the image can extend past its boundaries.

Improving performance when resizing Panel containers

When you apply a Resize effect to a Panel container, the measurement and layout algorithm for the effect executes repeatedly over the duration of the effect. When a Panel container has many children, the animation can be jerky because Flex cannot update the screen quickly enough. Also, resizing one Panel container often causes other Panel containers to resize.

To solve this problem, you can use the Resize effect's `hideChildrenTargets` property to hide the children of Panel containers while the Resize effect is playing. The value of the `hideChildrenTargets` property is an Array of Panel containers that should include the Panel containers that resize during the animation. Before the Resize effect plays, Flex iterates through the Array and hides the children of each of the specified Panel containers.

In the following example, the children of the panelOne and panelTwo Panel containers are hidden while the Panel containers resize:

```
<?xml version="1.0"?>
<!-- behaviors\PanelResize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Resize id="myResize" heightTo="300"
        hideChildrenTargets="{[panelOne, panelTwo]}"/>

    <mx:HBox>
        <mx:Panel id="panelOne" mouseDownEffect="{myResize}">
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
        </mx:Panel>
        <mx:Panel id="panelTwo" mouseDownEffect="{myResize}">
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
        </mx:Panel>
        <mx:Panel id="panelThree" mouseDownEffect="{myResize}">
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
            <mx:Button/>
        </mx:Panel>
    </mx:HBox>
</mx:Application>
```

For each Panel container in the `hideChildrenTargets` Array, the following effect triggers execute:

- `resizeStartEffect` Delivered before the Resize effect begins playing.
- `resizeEndEffect` Delivered after the Resize effect finishes playing.

If the `resizeStartEffect` trigger specifies an effect to play, the Resize effect is delayed until the effect finishes playing.

The default value for the Panel container's `resizeStartEffect` and `resizeEndEffect` triggers is `Dissolve`, which plays the Dissolve effect. For more information about the Dissolve effect, see "Available effects" on page 654.

To disable the Dissolve effect so that a Panel container's children are hidden immediately, you must set the value of the `resizeStartEffect` and `resizeEndEffect` triggers to `none`.

Setting UIComponent.cachePolicy on the effect target

An effect can use the bitmap caching feature in Flash Player to speed up animations. An effect typically uses bitmap caching when the target component's drawing does not change while the effect is playing.

For example, the Fade effect works by modifying the `alpha` property of the target component. Changing the `alpha` property does not change the way the target component is drawn on the screen. Therefore, caching the target component as a bitmap can speed up the performance of the effect. The Move effect modifies the `x` and `y` properties of the target component. Modifying the values of these properties does not alter the way the target component is drawn, so it can take advantage of bitmap caching.

Not all effects can use bitmap caching. Effects such as Zoom, Resize, and the wipe effects modify the target component in a way that alters the way it is drawn on the screen. The Zoom effect modifies the scale properties of the component, which changes its size. Caching the target component as a bitmap for such an effect would be counterproductive because the bitmap changes continuously while the effect plays.

The `UIComponent.cachePolicy` property controls the caching operation of a component during an effect. The `cachePolicy` property can have the following values:

CachePolicy.ON Specifies that the effect target is always cached.

CachePolicy.OFF Specifies that the effect target is never cached.

CachePolicy.AUTO Specifies that Flex determines whether the effect target should be cached. This is the default value. Flex uses the following rules to set the `cacheAsBitmap` property:

- When at least one effect that does not support bitmap caching is playing on a target, set the target's `cacheAsBitmap` property to `false`.
- When one or more effects that supports bitmap caching are playing on a target, set the target's `cacheAsBitmap` property to `true`.

Typically, you leave the `cachePolicy` property with its default value of `CachePolicy.AUTO`. However, you might want to set the property to `CachePolicy.OFF` because bitmap caching is interfering with your user interface, or because you know something about your application's behavior such that disabling bitmap caching will have a beneficial effect on it.

CHAPTER 18

Using Styles and Themes

18

Styles are useful for defining the look and feel (appearance) of your Adobe Flex applications. You can use them to change the appearance of a single component, or apply them across all components. This topic describes how to use styles, including the Cascading Style Sheet (CSS) syntax, in your applications. It also describes how to use themes.

Contents

About styles

You modify the appearance of Flex components through style properties. These properties can define the size of a font used in a Label control, or the background color used in the Tree control. In Flex, some styles are inherited from parent containers to their children, and across style types and classes. This means that you can define a style once, and then have that style apply to all controls of a single type or to a set of controls. In addition, you can override individual properties for each control at a local, component, or global level, giving you great flexibility in controlling the appearance of your applications.

This section introduces you to applying styles to controls. It also provides a primer for using Cascading Style Sheets (CSS), an overview of the style value formats (`Length`, `Color`, and `Time`), and describes style inheritance. Subsequent sections provide detailed information about different ways of applying styles in Flex.

Flex does not support controlling all aspects of component layout with CSS. Properties such as `x`, `y`, `width`, and `height` are properties, not styles, of the UIComponent class and, therefore, cannot be set in CSS. Other properties, such as `left`, `right`, `top`, and `bottom` are style properties and are used to manipulate a component's location in a container.

Using styles in Flex

There are many ways to apply styles in Flex. Some provide more granular control and can be approached programmatically. Others are not as flexible, but can require less computation. In Flex, you can apply styles to controls in several ways.

When applying styles, you must be aware of which properties your theme supports. The default theme in Flex does not support all style properties. For more information, see "About supported styles" on page 724.

External style sheets

Use CSS to apply styles to a document or across entire applications. You can point to a style sheet without invoking ActionScript. This is the most concise method of applying styles, but can also be the least flexible. Style sheets can define global styles that are inherited by all controls, or individual classes of styles that only certain controls use.

The following example applies the external style sheet myStyle.css to the current document:

```
<mx:Style source="myStyle.css"/>
```

For more information on using external style sheets, see "Using external style sheets" on page 726.

Flex includes a global style sheet, defaults.css, inside the framework.swc file. This file contains style definitions for the global class selector, and type selectors for most Flex components. For more information about defaults.css, see "About the default style sheet" on page 727.

Flex also includes several other style sheets that each have a unique look and feel. For more information, see "About the included theme files" on page 758.

Local style definitions

Use the <mx:Style> tag to define styles that apply to the current document and its children. You define styles in the <mx:Style> tag using CSS syntax and can define styles that apply to all instances of a control or to individual controls. The following example defines a new style and applies it to only the myButton control:

```
<?xml version="1.0"?>
<!-- styles/ClassSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 15;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
</mx:Application>
```

The following example defines a new style that applies to all instances of the Button class:

```
<?xml version="1.0"?>
<!-- styles/TypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        fontSize: 15;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" label="Click Here"/>
</mx:Application>
```

For more information on using local style definitions, see "Using local style definitions" on page 727.

StyleManager class

Use the StyleManager class to apply styles to all classes or all instances of specified classes. The following example sets the `fontSize` style to 15 and the `color` to `0x9933FF` on all Button controls:

```
<?xml version="1.0"?>
<!-- styles/StyleManagerExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">

  <mx:Script><![CDATA[
     public function initApp():void {
        StyleManager.getStyleDeclaration("Button").setStyle("fontSize",15);

StyleManager.getStyleDeclaration("Button").setStyle("color",0x9933FF);
     }
  ]]></mx:Script>

  <mx:Button id="myButton" label="Click Here"/>
</mx:Application>
```

You can also use the CSSStyleDeclaration object to build run time style sheets, and then apply them with the StyleManager's `setStyleDeclaration()` method.

For more information on using the StyleManager, see "Using the StyleManager class" on page 730.

getStyle() and setStyle() methods

Use the `setStyle()` and `getStyle()` methods to manipulate style properties on instances of controls. Using these methods to apply styles requires a greater amount of processing power on the client than using style sheets but provides more granular control over how styles are applied.

The following example sets the `fontSize` to 15 and the `color` to `0x9933FF` on only the myButton instance:

```
<?xml version="1.0"?>
<!-- styles/SetStyleExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">

  <mx:Script><![CDATA[
     public function initApp():void {
        myButton.setStyle("fontSize",15);
        myButton.setStyle("color",0x9933FF);
     }
  ]]></mx:Script>

  <mx:Button id="myButton" label="Click Here"/>
</mx:Application>
```

For more information on using the `getStyle()` and `setStyle()` methods, see "Using the setStyle() and getStyle() methods" on page 737.

Inline styles

Use attributes of MXML tags to apply style properties. These properties apply only to the instance of the control. This is the most efficient method of applying instance properties because no ActionScript code blocks or method calls are required.

The following example sets the `fontSize` to `15` and the `color` to `0x9933FF` on the myButton instance:

```
<?xml version="1.0"?>
<!-- styles/InlineExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Button id="myButton" color="0x9933FF" fontSize="15" label="Click
Here"/>
</mx:Application>
```

In an MXML tag, you must use the camel-case version of the style property. For example, you must use "`fontSize`" rather than "`font-size`" (the CSS convention) in the previous example. For more information on style property names, see "About property and selector names" on page 712.

As with other style properties, you can bind inline style properties to variables.

For more information on using inline styles, see "Using inline styles" on page 742.

Setting global styles

Most text and color styles, such as `fontSize` and `color`, are inheritable. When you apply an inheritable style to a container, all the children of that container inherit the value of that style property. If you set the `color` of a Panel container to green, all buttons in the Panel container are also green, unless those buttons override that color.

Many styles, however, are not inheritable. If you apply them to a parent container, only the container uses that style. Children of that container do not use the values of noninheritable styles.

By using global styles, you can apply noninheritable styles to all controls that do not explicitly override that style. Flex provides the following ways to apply styles globally:

- StyleManager `global` style
- CSS `global` selector

The StyleManager lets you apply styles to all controls using the `global` style. For more information on using the StyleManager class, see "Using the StyleManager class" on page 730.

You can also apply global styles using the `global` selector in your CSS style definitions. These are located either in external CSS style sheets or in an `<mx:Style>` tag. For more information, see "Using the global selector" on page 729.

About style value formats

Style properties can be of types String or Number. They can also be Arrays of these types. In addition to a type, style properties also have a format (`Length`, `Time`, or `Color`) that describes the valid values of the property. Some properties appear to be of type Boolean (taking a value of true or false), but these are Strings that are interpreted as Boolean values. This section describes these formats.

Length format

The `Length` format applies to any style property that takes a size value, such as the size of a font (or `fontSize`). `Length` is of type Number.

The `Length` type has the following syntax:

`length``[`*`length_unit`*`]`

The following example defines the `fontSize` property with a value of 20 pixels:

```
<?xml version="1.0"?>
<!-- styles/LengthFormat.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 20px;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
</mx:Application>
```

If you do not specify a unit, the unit is assumed to be pixels. The following example defines two styles with the same font size:

```
<?xml version="1.0"?>
<!-- styles/LengthFormat2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 20px;
        color: #9933FF;
     }
     .myOtherFontStyle {
        fontSize: 20;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
  <mx:Button id="myButton2" styleName="myOtherFontStyle" label="Click
Here"/>
</mx:Application>
```

NOTE Spaces are not allowed between the length value and the unit.

The following table describes the supported `length` units:

Unit	Scale	Description
px	Relative	Pixels.
in	Absolute	Inches.
cm	Absolute	Centimeters.
mm	Absolute	Millimeters.
pt	Absolute	Points.
pc	Absolute	Picas.

Flex does not support the em and ex units. You can convert these to px units by using the following scales:

1em = 10.06667px

1ex = 6px

In Flex, all lengths are converted to pixels prior to being displayed. In this conversion, Flex assumes that an inch equals 72 pixels. All other lengths are based on that assumption. For example, 1 cm is equal to 1/2.54 of an inch. To get the number of pixels in 1 cm, multiply 1 by 72, and divide by 2.54.

When you use inline styles, Flex ignores units and uses pixels as the default.

The `fontSize` style property allows a set of keywords in addition to numbered units. You can use the following keywords when setting the `fontSize` style property. The exact sizes are defined by the client browser.

- `xx-small`
- `x-small`
- `small`
- `medium`
- `large`
- `x-large`
- `xx-large`

The following example class selector defines the `fontSize` as `x-small`:

```
<?xml version="1.0"?>
<!-- styles/SmallFont.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .smallFont {
        fontFamily: Arial, Helvetica, "_sans";
        fontSize: x-small;
        fontStyle: oblique;
     }
  </mx:Style>

  <mx:Button id="myButton" styleName="smallFont" label="Click Here"/>

</mx:Application>
```

Time format

You use the `Time` format for component properties that move or have built-in effects, such as the ComboBox component when it drops down and pops up. The `Time` format is of type Number and is represented in milliseconds. Do not specify the units when entering a value in the `Time` format.

The following example sets the `selectionDuration` style property of the myTree control to 100 milliseconds:

```
<?xml version="1.0"?>
<!-- styles/SetStyleExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">

  <mx:Script><![CDATA[
     public function initApp():void {
        myTree.setStyle("selectionDuration", 100);
     }
  ]]></mx:Script>

    <mx:XMLList id="treeData">
        <node label="Mail Box">
            <node label="Inbox">
                <node label="Marketing"/>
                <node label="Product Management"/>
                <node label="Personal"/>
            </node>
            <node label="Outbox">
                <node label="Professional"/>
                <node label="Personal"/>
            </node>
            <node label="Spam"/>
        <node label="Sent"/>
     </node>
  </mx:XMLList>

  <mx:Panel title="Tree Control Example" width="100%">
        <mx:Tree id="myTree" width="100%" labelField="@label"
dataProvider="{treeData}"/>
  </mx:Panel>
</mx:Application>
```

Color format

You define `Color` in several formats. You can use most of the formats only in the CSS style definitions. The following table describes the recognized `Color` formats for a style property:

Format	Description
hexadecimal	Hexadecimal colors are represented by a six-digit code preceded by either a zero and small x (`0x`) or a pound sign (#). The range of valid values is `0x000000` to `0xFFFFFF` (or `#000000` to `#FFFFFF`). You use the `0x` prefix when defining colors in calls to the `setStyle()` method and in MXML tags. You use the # prefix in CSS style sheets and in `<mx:Style>` tag blocks.
RGB	RGB colors are a mixture of the colors red, green, and blue, and are represented in percentages of the color's saturation. The format for setting RGB colors is `color:rgb(x%, y%, z%)`, where the first value is the percentage of red saturation, the second value is the percentage of green saturation, and the third value is the percentage of blue saturation. You can use the RGB format only in style sheet definitions.
VGA color names	VGA color names are a set of 16 basic colors supported by all browsers that support CSS. The available color names are `Aqua`, `Black`, `Blue`, `Fuchsia`, `Gray`, `Green`, `Lime`, `Maroon`, `Navy`, `Olive`, `Purple`, `Red`, `Silver`, `Teal`, `White`, `Yellow`. You can use the VGA color names format in style sheet definitions and inline style declarations. VGA color names are not case-sensitive.

Color formats are of type Number. When you specify a format such as a VGA color name, Flex converts that String to a Number.

CSS style definitions and the `<mx:Style>` tag support the following color value formats:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myStyle {
        themeColor: #6666CC;    /* CSS hexadecimal format */
        color: Blue;        /* VGA color name */
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myStyle" label="Click Here"/>
</mx:Application>
```

You can use the following color value formats when setting the styles inline or using the `setStyle()` method:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatStyleManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     public function initApp():void {
        StyleManager.getStyleDeclaration("Button").
            setStyle("themeColor",0x6666CC);
        StyleManager.getStyleDeclaration("Button").
            setStyle("color","Blue");
     }

     public function changeStyles(e:Event):void {
        e.currentTarget.setStyle("themeColor", 0xFF0099);
        e.currentTarget.setStyle("color", "Red");
     }
  ]]></mx:Script>
  <mx:Button id="myButton"
        label="Click Here"
        click="changeStyles(event)"
  />
</mx:Application>
```

Using Arrays for style properties

Some controls accept arrays of colors. For example, the Tree control's `depthColors` style property can use a different background color for each level in the tree. To assign colors to a property in an Array, add the items in a comma-separated list to the property's definition. The index is assigned to each entry in the order that it appears in the list.

The following example defines Arrays of colors for properties of the `Tree` type selector:

```
<?xml version="1.0"?>
<!-- styles/ArraysOfColors.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Tree {
        depthColors: #FFCC33, #FFCC99, #CC9900;
        alternatingItemColors: red, green;
     }
  </mx:Style>

  <mx:XMLList id="treeData">
     <node label="Mail Box">
        <node label="Inbox">
           <node label="Marketing"/>
           <node label="Product Management"/>
           <node label="Personal"/>
        </node>
        <node label="Outbox">
           <node label="Professional"/>
           <node label="Personal"/>
        </node>
        <node label="Spam"/>
        <node label="Sent"/>
     </node>
  </mx:XMLList>

  <mx:Panel title="Tree Control Example" width="100%">
        <mx:Tree id="myTree" width="100%" labelField="@label"
dataProvider="{treeData}"/>
  </mx:Panel>
</mx:Application>
```

In this example, only `depthColors` will be seen. The `alternatingItemColors` property is only visible if `depthColors` is not set. Both are presented here for illustrative purposes only.

You can define the Array in ActionScript by using a comma-separated list of values surrounded by braces, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/SetStyleArray.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     public function initApp():void {
        myTree.setStyle("depthColors",[0xFFCC33, 0xFFCC99, 0xCC9900]);
        myTree.setStyle("alternatingItemColors",["red", "green"]);
     }
  ]]></mx:Script>
  <mx:XMLList id="treeData">
     <node label="Mail Box">
        <node label="Inbox">
           <node label="Marketing"/>
           <node label="Product Management"/>
           <node label="Personal"/>
        </node>
        <node label="Outbox">
           <node label="Professional"/>
           <node label="Personal"/>
        </node>
        <node label="Spam"/>
        <node label="Sent"/>
     </node>
  </mx:XMLList>

  <mx:Panel title="Tree Control Example" width="100%">
        <mx:Tree id="myTree"
           width="100%"
           labelField="@label"
           dataProvider="{treeData}"
        />

        <mx:Tree id="myOtherTree"
           width="100%"
           labelField="@label"
           dataProvider="{treeData}"
           depthColors="[0xFFCC33, 0xFFCC99, 0xCC9900]"
           alternatingItemColors="['red', 'green']"
        />
  </mx:Panel>
</mx:Application>
```

This example also shows that you can set the properties that use Arrays inline.

Finally, you can set the values of the Array in MXML syntax and apply those values inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ArrayOfColorsMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Array id="myDepthColors">
     <mx:Object>0xFFCC33</mx:Object>
     <mx:Object>0xFFCC99</mx:Object>
     <mx:Object>0xCC9900</mx:Object>
  </mx:Array>

  <mx:Array id="myAlternatingRowColors">
     <mx:Object>red</mx:Object>
     <mx:Object>green</mx:Object>
  </mx:Array>

  <mx:XMLList id="treeData">
     <node label="Mail Box">
        <node label="Inbox">
           <node label="Marketing"/>
           <node label="Product Management"/>
           <node label="Personal"/>
        </node>
        <node label="Outbox">
           <node label="Professional"/>
           <node label="Personal"/>
        </node>
        <node label="Spam"/>
        <node label="Sent"/>
     </node>
  </mx:XMLList>

  <mx:Panel title="Tree Control Example" width="100%">
        <mx:Tree id="myTree"
           width="100%"
           labelField="@label"
           dataProvider="{treeData}"
           depthColors="{myDepthColors}"
           alternatingItemColors="{myAlternatingRowColors}"
        />
  </mx:Panel>
</mx:Application>
```

Using Cascading Style Sheets

Cascading Style Sheets (CSS) are a standard mechanism for declaring text styles in HTML and most scripting languages. A style sheet is a collection of formatting rules for types of components or classes that include sets of components. Flex supports the use of CSS syntax and styles to apply styles to Flex components.

In CSS syntax, each declaration associates a style name, or *selector*, with one or more style properties and their values. You define multiple style properties in each selector by separating each property with a semicolon. For example, the following style defines a selector named `myFontStyle`:

```
<?xml version="1.0"?>
<!-- styles/ClassSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 15;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
</mx:Application>
```

In this example, `myFontStyle` defines a new class of styles, so it is called a *class selector*. In the markup, you can explicitly apply the `myFontStyle` style to a control or class of controls.

A *type selector* implicitly applies itself to all components of a particular type, as well as all subclasses of that type.

The following example defines a type selector named `Button`:

```
<?xml version="1.0"?>
<!-- styles/TypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        fontSize: 15;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" label="Click Here"/>
</mx:Application>
```

Flex applies this style to all Button controls, and all subclasses of Button controls. If you define a type selector on a container, that style applies to all children of that container if the style is an inheriting style.

When determining styles for a new component instance, Flex examines all the parent classes looking for type selectors. Flex applies settings in all type selectors, not just the exact match. For example, suppose that class MyButton extends Button. For an instance of MyButton, Flex first checks for a MyButton type selector. Flex applies styles in the MyButton type selector, and then checks for a Button type selector. Flex applies styles in the Button selector, and then checks for a UIComponent type selector. Flex stops at UIComponent. Flex does not continue up the parent chain past UIComponent because Flex does not support type selectors for Sprite (the parent of UIComponent) or any of Sprite's parent classes, up to the base Object class.

NOTE

The names of class selectors cannot include hyphens in Flex. If you use a hyphenated class selector name, such as my-class-selector, Flex ignores the style.

You can programmatically define new class and type selectors using the StyleManager class. For more information, see "Using the StyleManager class" on page 730.

About property and selector names

When applying style properties with CSS in a `<mx:Style>` block or in an external style sheet, the best practice is to use camel-case for the style property, as in `fontWeight` and `fontFamily`.

To make development easier, Flex supports both the camel-case and hyphenated syntax in style sheets, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/CamelCase.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 15;
     }
     .myOtherFontStyle {
        font-size: 15;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
  <mx:Button id="myButton2" styleName="myOtherFontStyle" label="Click
Here"/>
  <mx:Button id="myButton3" fontSize="15" label="Click Here"/>
</mx:Application>
```

In ActionScript or an MXML tag, you cannot use hyphenated style property names, so you must use the camel-case version of the style property. For the style name in a style sheet, you cannot use a hyphenated name, as the following example shows:

```
.myClass { ... } /* Valid style name */
.my-class { ... } /* Not a valid style name */
```

About inheritance in CSS

Some style properties are inherited. If you set an inheritable style property on a parent container, its children inherit that style property. For example, if you define `fontFamily` as `Times` for a Panel container, all children of that container will also use `Times` for `fontFamily`, unless they override that property. If you set a noninheritable style such as `textDecoration` on a parent container, only the parent container and not its children use that style. For more information on inheritable style properties, see "About style inheritance" on page 721.

In general, color and text styles are inheritable, regardless of how they are set (by using style sheets or the `setStyle()` method). All other styles are not inheritable unless otherwise noted.

The following are exceptions to the rules of inheritance:

- If you use the `global` selector in a CSS style definition, Flex applies those style properties to all controls, regardless of whether the properties are inheritable. For more information about the `global` selector, see "Using the global selector" on page 729.
- The values set in type selectors apply to the target class as well as its subclasses, even if the style properties are not inheritable. For example, Flex applies all styles in a VBox type selector, as well as styles in a Container type selector, to VBox containers.

CSS differences

There are some major differences in Flex between support of CSS and the CSS specification:

- Flex supports a subset of the style properties that are available in CSS. Flex controls also have unique style properties that are not defined by the CSS specification. For a list of styles that you can apply to your Flex controls, see "Supported CSS properties" on page 719.
- Flex controls only support styles that are defined by the current theme. If a theme does not use a particular style, applying that style to a control or group of controls has no effect. For example, the default theme, Halo Aeon, does not support styles such as `symbolColor` and `symbolBackgroundColor`. For more information on themes, see "About themes" on page 756.
- Flex style sheets can define skins for controls using the Embed keyword. For more information, see Chapter 20, "Using Skins," on page 805.

About class selectors

Class selectors define a set of styles (or a class) that you can apply to any component. You define the style class, and then point to the style class using the `styleName` property of the component's MXML tag. All Flex components that are a subclass of the UIComponent class support the `styleName` property.

The following example defines a new style `myFontStyle` and applies that style to a Button component by assigning the Button to the `myFontStyle` style class:

```
<?xml version="1.0"?>
<!-- styles/ClassSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 15;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
</mx:Application>
```

Class selector names must start with a period when you access them with the `getStyleDeclaration()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ClassSelectorStyleManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 15;
        color: #9933FF;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     public function changeStyles(e:Event):void {

StyleManager.getStyleDeclaration('.myFontStyle').setStyle('color',0x3399CC)
;
     }
  ]]></mx:Script>

  <mx:Button id="myButton" label="Click Here" styleName="myFontStyle"
click="changeStyles(event)"/>

</mx:Application>
```

You do not precede the class selector with a period when you use the `styleName` property for inline styles.

About type selectors

Type selectors assign styles to all components of a particular type. When you define a type selector, you are not required to explicitly apply that style. Instead, Flex applies the style to all classes of that type. Flex also applies the style properties defined by a type selector to all subclasses of that type.

The following example shows a type selector for Button controls:

```
<?xml version="1.0"?>
<!-- styles/TypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        fontSize: 15;
        color: #9933FF;
     }
  </mx:Style>
  <mx:Button id="myButton" label="Click Here"/>
</mx:Application>
```

In this example, Flex applies the `color` style to all Button controls in the current document, and all Button controls in all the child documents. In addition, Flex applies the `color` style to all subclasses of Button.

You can set the same style declaration for multiple component types by using a comma-separated list of components. The following example defines style information for all Button, TextInput, and Label components:

```
<?xml version="1.0"?>
<!-- styles/MultipleTypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button, TextInput, Label {
        fontStyle: italic;
        fontSize: 24;
     }
  </mx:Style>

  <mx:Button id="myButton" label="Click Here"/>
  <mx:Label id="l1" text="My Label"/>
  <mx:TextInput id="ti1" text="Input text here"/>
</mx:Application>
```

You can use multiple type selectors of the same name at different levels to set different style properties. In an external CSS file, you can set all Label components to use the Blue color for the fonts, as the following example shows:

```
/* assets/SimpleTypeSelector.css */
Button {
    fontStyle: italic;
    color: #99FF00;
}
```

Then, in a local style declaration, you can set all Labels to use the font size 10, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/TypeSelectorWithExternalCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style source="../assets/SimpleTypeSelector.css"/>

  <mx:Style>
     Button {
        fontSize: 15;
     }
  </mx:Style>
  <mx:Button id="myButton" label="Click Here"/>
</mx:Application>
```

The local style declaration does not interfere with external style declarations. Flex applies only the style properties that you specified. The result of this example is that Label controls that are children of the current document use Blue for the color and 10 for the font size.

All styles are shared across all documents in an application and across all applications that are loaded inside the same application. For example, if you load two SWF files inside separate tabs in a TabNavigator container, both SWF files share the external style definitions.

Using compound selectors

You can mix class and type selectors to create a component that has styles based on compound style declarations. For example, you can define the color in a class selector and the font size in a type selector, and then apply both to the component:

```
<?xml version="1.0"?>
<!-- styles/CompoundSelectors.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     Label {
        fontSize: 10pt;
     }
     .myLabel {
        color: Blue;
     }
  </mx:Style>

  <mx:Label styleName="myLabel" text="This label is 10pt Blue"/>

</mx:Application>
```

If you later remove one of the selectors (for example, call the StyleManager's `clearStyleDeclaration()` method on the type or class selector), the other selector's style settings remain.

About selector precedence

Class selectors take precedence over type selectors. In the following example, the text for the first button (with the class selector) is red, and the text of the second button (with the implicit type selector) is yellow:

```
<?xml version="1.0"?>
<!-- styles/SelectorPrecendence.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myclass {
        color: Red;
     }
     Button {
        fontSize: 10pt;
        color: Yellow;
     }
  </mx:Style>
  <mx:VBox width="500" height="200">
     <mx:Button styleName="myclass" label="I am red"/>
     <mx:Button label="I am yellow"/>
  </mx:VBox>
</mx:Application>
```

The font size of both buttons is 10. When a class selector overrides a type selector, it does not override all values, just those that are explicitly defined.

Type selectors apply to a particular class, as well as its subclasses and child components. In the following example, the `color` property for a VBox control is `blue`. This means that the `color` property for the Button and Label controls, which are direct children of the VBox control, is blue.

```
<?xml version="1.0"?>
<!-- styles/BasicInheritance.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     VBox {
        color:blue
     }
  </mx:Style>

  <mx:VBox width="500" height="200">
     <mx:Label text="This is a label"/>
     <mx:Button label="Click Me"/>
  </mx:VBox>
</mx:Application>
```

If the same style property is applied in multiple type selectors that apply to a class, the closest type to that class takes precedence. For example, the VBox class is a subclass of Box, which is a subclass of Container. If there were Box and Container type selectors rather than a VBox type selector, then the value of the VBox control's `color` property would come from the Box type selector rather than the Container type selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ContainerInheritance.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     Container {
        color:red
     }
     Box {
        color:green
     }
  </mx:Style>

  <mx:VBox width="500" height="200">
     <mx:Label text="This is a green label"/>
     <mx:Button label="Click this green button"/>
  </mx:VBox>
</mx:Application>
```

Not all style properties are inheritable. For more information, see "About style inheritance" on page 721.

Supported CSS properties

Flex supports the following subset of the CSS style properties as defined by the CSS specification:

- `color`
- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `paddingBottom`
- `paddingLeft`
- `paddingRight`
- `paddingTop`
- `textAlign`
- `textDecoration`

- `textIndent`

Flex also supports properties that you can define by using CSS syntax and inheritance rules, but the properties are not part of the CSS property library. For a list of style properties for each control, view that control's entry in *Adobe Flex 2 Language Reference*.

Embedding resources in style sheets

You can use embedded resources in your `<mx:Style>` blocks. This is useful for style properties such as `backgroundImage`, which you can apply to an embedded resource such as an image file. The following example embeds an image as myImage. The style declaration references this resource:

```
<?xml version="1.0"?>
<!-- styles/EmbedInCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .style1 {
        backgroundImage: Embed("../assets/bird.gif");
     }
  </mx:Style>
  <mx:HBox>
     <mx:TextArea width="200" height="200" styleName="style1"/>
  </mx:HBox>
</mx:Application>
```

For graphical skins, you use the `Embed` statement directly in the style sheet, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        overSkin: Embed("../assets/orb_over_skin.gif");
        upSkin: Embed("../assets/orb_up_skin.gif");
        downSkin: Embed("../assets/orb_down_skin.gif");
     }
  </mx:Style>
  <mx:Button id="b1" label="Click Me"/>
</mx:Application>
```

For programmatic skins, you use the `ClassReference` statement, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatesSkin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        overSkin: ClassReference("ButtonStatesSkin");
        upSkin: ClassReference("ButtonStatesSkin");
        downSkin: ClassReference("ButtonStatesSkin");
     }
  </mx:Style>
  <mx:Button id="b1" label="Click Me"/>
</mx:Application>
```

For more information about using Embed, see Chapter 30, "Embedding Assets," on page 1113. For more information about skinning, see Chapter 20, "Using Skins," on page 805.

About style inheritance

If you define a style in only one place in a document, Flex uses that definition to set a property's value. However, an application can have several style sheets, local style definitions, external style properties, and style properties set directly on component instances. In such a situation, Flex determines the value of a property by looking for its definition in all these places in a specific order.

Lower-level styles take precedence over higher-level or external styles. If you set a style on an instance, and then set the style globally, the global style does not override the local style, even if you set it after you set the local style.

Style inheritance order

The order in which Flex looks for styles is important to understand so that you know which style properties apply to which controls.

Flex looks for a style property that was set inline on the component instance. If no style was set on the instance using an inline style, Flex checks if a style was set using an instance's `setStyle()` method. If it did not directly set the style on the instance, Flex examines the `styleName` property of the instance to see if a style declaration is assigned to it.

If you did not assign the `styleName` property to a style declaration, Flex looks for the property on type selector style declarations. If there are no type selector declarations, Flex checks the `global` selector. If all of these checks fail, the property is undefined, and Flex applies the default style.

In the early stages of checking for a style, Flex also examines the control's parent container for style settings. If the style property is not defined and the property is inheritable, Flex looks for the property on the instance's parent container. If the property isn't defined on the parent container, Flex checks the parent's parent, and so on. If the property is not inheritable, Flex ignores parent container style settings.

The order of precedence for style properties, from first to last, is as follows:

- Inline
- Class selector
- Type selectors (most immediate class takes precedence when multiple selectors apply the same style property)
- Parent chain (inheriting styles only)
- `global` selector

If you later call the `setStyle()` method on a component instance, that method takes precedence over all style settings, including inline.

Style definitions in `<mx:Style>` tags, external style sheets, and the defaults.css style sheet follow an order of precedence. The same style definition in defaults.css is overridden by an external style sheet that is specified by an `<mx:Style source="stylesheet"/>` tag, which is overridden by a style definition within an `<mx:Style>` tag.

The following example defines a type selector for Panel that sets the `fontFamily` property to `Times` and the `fontSize` property to 24. As a result, all controls inside the Panel container, as well as all subclasses such as Button and TextArea, inherit those styles. However, button2 overrides the inherited styles by defining them inline. When the application renders, button2 uses Arial for the font and 12 for the font size.

```
<?xml version="1.0"?>
<!-- skins/MoreContainerInheritance.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
  Panel {
     fontFamily: Times, "_serif";
     fontSize: 24;
  }
  </mx:Style>
  <mx:Panel title="My Panel">
     <mx:Button id="button1" label="Button 1"/>
     <mx:Button id="button2" label="Button 2" fontFamily="Arial"
fontSize="12"/>
     <mx:TextArea text="Flex has is own set of style properties which are
        extensible so you can add to that list when you create a custom
        component." width="425" height="400"/>
  </mx:Panel>
</mx:Application>
```

A potentially unexpected consequence of type selector style inheritance is when a Flex component contains subcomponents. Subcomponents inherit the style properties that are set in their type selectors, just as if they were stand-alone components.

For example, the TextInput class is used as a subcomponent in the ComboBox, NumericStepper, and RichTextEditor controls. So, if you define a type selector for the TextInput type, the areas of the ComboBox, NumericStepper, and RichTextEditor controls that use the TextInput class also reflect those styles.

The following example sets the `fontSize` style to 15 in the TextInput type selector. As a result, the text in the RichTextEditor's font list drop-down list, the NumericStepper's text region, and the text in the ComboBox's drop-down list, font size drop-down list, and URL field are also size 15:

```
<?xml version="1.0"?>
<!-- skins/SubComponentInheritance.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     TextInput {
        fontSize:15;
     }
  </mx:Style>

  <mx:RichTextEditor/>

  <mx:ComboBox>
     <mx:dataProvider>
        <mx:String>2005</mx:String>
        <mx:String>2006</mx:String>
        <mx:String>2007</mx:String>
     </mx:dataProvider>

  </mx:ComboBox>
  <mx:NumericStepper/>

</mx:Application>
```

Inheritance exceptions

Not all styles are inheritable, and not all styles are supported by all components and themes. In general, color and text styles are inheritable, regardless of how they are set (using CSS or style properties). All other styles are not inheritable unless otherwise noted.

A style is inherited only if it meets the following conditions:

- The style is supported by the theme. For a list of styles supported by the default Flex theme, see "About supported styles" on page 724.

- The style is inheritable. You can see a list of inherited style for each control by viewing that control's entry in *Adobe Flex 2 Language Reference*. You can also determine if a style is inheritable using the static `isInheritingStyle()` or `isInheritingTextFormatStyle()` methods on the StyleManager class.
- The style is supported by the control. For information about which controls support which styles, see the control's description in *Adobe Flex 2 Language Reference*.
- The style is set on the control's parent container or the container's parent. A style doesn't get inherited from another class, unless that class is a parent container of the control, or a parent container of the control's parent container. (The exception to this condition is if you use type selectors to apply the style property. In that case, Flex applies properties of the class's type selector, as well as any properties set in the base class's type selector.)
- The style is not overridden at a lower level. For example, if you define a style type selector (such as `Button { color:red }`), but then set an instance property on a control (such as `<mx:Button color="blue"/>`, the type selector style will not override the style instance property even if the style is inheritable.

You can apply noninheritable styles to all controls using the `global` selector. For more information, see "Using the global selector" on page 729.

About supported styles

All themes support the inheritable and noninheritable text styles, but not all styles are supported by all themes. If you try to set a style property on a control but the current theme does not support that style, Flex does not apply the style.

Some styles are only used by skins in the theme, while others are used by the component code itself. The display text of components is not skinnable, so support for text styles is theme-independent.

Themes		Supported styles	

All	Inheritable	`color`
		`fontFamily`
		`fontSize`
		`fontStyle`
			`fontWeight`
			`textAlign`
			`textIndent`

All	Noninheritable	`paddingLeft`
		`paddingRight`
		`textDecoration`	

Themes		Supported styles	
All	Component - defined	disabledColor headerHeight horizontalAlign horizontalGap leading paddingBottom paddingLeft paddingRight paddingTop	rollOverColor selectionColor tabHeight tabWidth textRollOverColor textSelectedColor verticalAlign verticalGap
Halo Aeon	Component-dependent and all text styles	backgroundGradientAlphas backgroundGradientColors borderAlpha borderColor borderSides borderThickness cornerRadius dateHeaderColor dateRollOverColor dropShadowEnabled fillAlphas fillColors	footerColors headerColors highlightAlphas highlightColor roundedBottomCorners selectedDateColor shadowDirection shadowDistance strokeWidth themeColor todayColor

For more information about themes, "About themes" on page 756.

About the themeColor property

Many assets in the default Halo theme support a property called `themeColor`. You can set this property on the Application tag, and the color is applied throughout the Flex application on component assets, such as the Button control's border, the headers of an Accordion control, and the default shading of a ToolTip control's background.

In addition to color values such as 0xCCCCCC (for silver) or 0x0066FF (for blue), the following values for the `themeColor` property are valid:

- `haloOrange`
- `haloBlue`
- `haloSilver`
- `haloGreen`

The default value is `haloBlue`. The following example sets the value of `themeColor` to `haloOrange`:

```
<?xml version="1.0"?>
<!-- styles/ThemeColorExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
themeColor="haloOrange">

  <mx:Button id="myButton" label="Click Me" toolTip="Click me"/>

</mx:Application>
```

Using external style sheets

Flex supports external CSS style sheets. You can declare the location of a local style sheet or use the external style sheet to define the styles that all applications use. To apply a style sheet to the current document and its child documents, use the `source` property of the `<mx:Style>` tag.

NOTE

You should try to limit the number of style sheets used in an application, and set the style sheet only at the top-level document in the application (the document that contains the `<mx:Application>` tag). If you set a style sheet in a child document, unexpected results can occur.

The following example points to the MyStyleSheet.css file in the *flex_app_root*/assets directory:

```
<?xml version="1.0"?>
<!-- styles/ExternalCSSExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style source="../assets/SimpleTypeSelector.css"/>

  <mx:Button id="myButton" label="Click Here"/>
</mx:Application>
```

The value of the `source` property is the URL of a file that contains style declarations. When you use the `source` property, the contents of *that* `<mx:Style>` tag must be empty. You can use additional `<mx:Style>` tags to define other styles. Do not add `<mx:Style>` tags to your included file.

The external style sheet file can contain both type and class selectors.

You can also compile CSS files into SWF files and load them at run time. For more information, see "Loading style sheets at run time" on page 744.

About the default style sheet

Flex includes a default style sheet that is used across all applications. This style sheet applies a consistent style across all Flex components. This file is defaults.css and is located inside the framework.swc file in the /frameworks/libs directory. The programmatic skin classes that it embeds are in the mx.skins.halo package. The graphical skins that is uses are also in the framework.swc file.

This default style sheet makes up the Halo theme. For more information about themes, see "About themes" on page 756.

Flex implicitly loads the defaults.css file and applies it to the Flex application during compilation. You can explicitly point to another file for the default styles by using the `defaults-css-url` compiler option. You can also rename the defaults.css file or remove it from the framework.swc file to disable it.

The defaults.css file defines the look and feel for all Flex components. If you apply additional themes or CSS files to your application, Flex still uses the styles in defaults.css, but only for the components that your custom styles do not override. To completely eliminate the default theme from Flex, you must remove or override all styles defined in defaults.css.

Flex also includes other style sheets that let you apply a theme quickly and easily. For more information, see "About the included theme files" on page 758.

Using local style definitions

The `<mx:Style>` tag contains style sheet definitions that adhere to the CSS 2.0 syntax. These definitions apply to the current document and all children of the current document. The `<mx:Style>` tag uses the following syntax to define local styles:

```
<mx:Style>
  selector_name {
    style_property: value;
    [...]
  }
</mx:Style>
```

The following example defines a class and a type selector in the `<mx:Style>` tag:

```
<?xml version="1.0"?>
<!-- styles/CompoundLocalStyle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 15;
        color: #9933FF;
     }

     Button {
        fontStyle: italic;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
</mx:Application>
```

Using the Application type selector

The Application container is the top-most container in a Flex application. Styles defined on the Application type selector that are inheritable are inherited by all of the container's children as well as the container's subclasses. Styles that are not inheritable are only applied to the Application container itself and not its children.

Styles applied with the Application type selector are not inherited by the Application object's children if those styles are noninheritable. To use CSS to apply a noninheritable style globally, you can use the `global` selector. For more information, see "Using the global selector" on page 729.

When you define the styles for the Application type selector, you are not required to declare a style for each component, because the components are children of these classes and inherit the Application type selector styles.

Use the following syntax to define styles for the Application type selector:

```
<mx:Style>
  Application { style_definition }
</mx:Style>
```

You can use the Application type selector to set the background image and other display settings that define the way the Flex application appears in a browser. The following sample Application style definition aligns the application file to the left, removes margins, and sets the background image to be empty:

```
<?xml version="1.0"?>
<!-- styles/ApplicationTypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Application {
        paddingLeft: 0px;
        paddingRight: 0px;
        paddingTop: 0px;
        paddingBottom: 0px;
        horizontalAlign: "left";
       backgroundImage: " "; /* The empty string sets the image to nothing.
*/
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
</mx:Application>
```

When the background image is set to the empty string, Flex does not draw the default gray gradient.

You can programmatically define values in the Application type selector using the StyleManager class. For more information, see "Using the StyleManager class" on page 730.

Using the global selector

Flex includes a `global` selector that you can use to apply styles to all controls. Properties defined by a `global` selector apply to every control unless that control explicitly overrides it. Because the `global` selector is like a type selector, you do not preface its definition with a period in CSS.

The following example defines `fontSize`, an inheritable property, and `textDecoration`, a noninheriting property, to the `global` selector:

```
<?xml version="1.0"?>
<!-- styles/GlobalTypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     global {
        fontSize:22;
        textDecoration: underline;
     }
  </mx:Style>
  <mx:Button id="myButton" label="Click Here"/>
  <mx:Label id="myLabel" text="This is a label"/>
</mx:Application>
```

You can also use the `getStyleDeclaration()` method to apply the styles with the global selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/GlobalTypeSelectorAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp(event)">
  <mx:Script><![CDATA[
     public function initApp(e:Event):void {
        StyleManager.getStyleDeclaration("global").
            setStyle("fontSize", 22);
        StyleManager.getStyleDeclaration("global").
            setStyle("textDecoration", "underline");
     }
  ]]></mx:Script>

  <mx:Button id="myButton" label="Click Here"/>
  <mx:Label id="myLabel" text="This is a label"/>
</mx:Application>
```

Class selectors, type selectors, and inline styles all override the `global` selector.

Using the StyleManager class

The StyleManager class lets you access class selectors and type selectors in ActionScript. It also lets you apply inheritable and noninheritable properties globally. Using the StyleManager, you can define new CSS style declarations and apply them to controls in your Flex applications.

To set a value using the StyleManager, use the following syntax:

```
mx.styles.StyleManager.getStyleDeclaration(style_name).setStyle("property",
  value);
```

The *style_name* can be the literal `global`, a type selector such as Button or TextArea, or a class selector that you define in either the `<mx:Style>` tag or an external style sheet. Global styles apply to every object that does not explicitly override them.

The `getStyleDeclaration()` method is useful if you apply a noninheritable style, such as `textDecoration`, to many classes at one time. This property refers to an object of type CSSStyleDeclaration. Type selectors and external style sheets are assumed to already be of type CSSStyleDeclaration. Flex internally converts class selectors that you define to this type of object.

The following examples illustrate applying style properties to the `Button`, `myStyle`, and `global` style names:

```
<?xml version="1.0"?>
<!-- styles/USingStyleManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp(event)">
  <mx:Style>
     .myStyle {
        color: red;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     import mx.styles.StyleManager;
     public function initApp(e:Event):void {
        // Type selector; applies to all Buttons and subclasses
        // of Button.
        StyleManager.getStyleDeclaration("Button").
            setStyle("fontSize",24);

        // Class selector; applies to controls using the style
        // named myStyle. Note that class selectors must be prefixed
        // with a period.
        StyleManager.getStyleDeclaration(".myStyle").
            setStyle("color",0xCC66CC);

        // Global style: applies to all controls.
        StyleManager.getStyleDeclaration("global").
            setStyle("fontStyle","italic");
     }
  ]]></mx:Script>

  <mx:Button id="myButton"
        label="Click Here"
        styleName="myStyle"
  />
  <mx:Label id="myLabel"
        text="This is a label"
        styleName="myStyle"
  />

</mx:Application>
```

NOTE

If you set either an inheritable or noninheritable style to the `global` style, Flex applies it to all controls, regardless of their location in the hierarchy.

You can create CSS style declarations by using ActionScript with the CSSStyleDeclaration class. This lets you create and edit style sheets at run time and apply them to classes in your Flex applications. To change the definition of the styles or to apply them during run time, you use the `setStyle()` method.

The StyleManager also includes a `setStyleDeclaration()` method that lets you apply a CSSStyleDeclaration object as a selector, so you can apply a style sheet to all components of a type. The selector can be a class or type selector.

The following example creates a new CSSStyleDeclaration object, applies several style properties to the object, and then applies the new style to all Button controls with the StyleManager's `setStyleDeclaration()` method:

```
<?xml version="1.0"?>
<!-- styles/StyleDeclarationTypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.styles.StyleManager;

     private var myDynStyle:CSSStyleDeclaration;

     private function initApp():void {
        myDynStyle = new CSSStyleDeclaration('myDynStyle');

        myDynStyle.setStyle('color', 'blue');
        myDynStyle.setStyle('fontFamily', 'georgia');
        myDynStyle.setStyle('themeColor', 'green');
        myDynStyle.setStyle('fontSize', 24);

        // Apply the new style to all Buttons. By using a type
        // selector, this CSSStyleDeclaration object will replace
        // all style properties, causing potentially unwanted
        // results.
        StyleManager.setStyleDeclaration("Button",myDynStyle, true);
     }
  ]]></mx:Script>

  <mx:Button id="myButton" label="Click Me"/>

</mx:Application>
```

When you set a new CSSStyleDeclaration on a type selector, you are replacing the entire existing type selector with your own selector. All style properties that you do not explicitly define in the new CSSStyleDeclaration are set to null. This can remove skins, margins, and other properties that are defined in the defaults.css file or other style sheet that you may have applied already.

To avoid nullifying all style properties, you can use a class selector to apply the new CSSStyleDeclaration object. Because a class selector's style properties do not replace existing type selector properties, the components maintain their default settings, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/StyleDeclarationClassSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.styles.StyleManager;

     private var myDynStyle:CSSStyleDeclaration;

     private function initApp():void {
        myDynStyle = new CSSStyleDeclaration('myDynStyle');

        myDynStyle.setStyle('color', 'blue');
        myDynStyle.setStyle('fontFamily', 'georgia');
        myDynStyle.setStyle('themeColor', 'green');
        myDynStyle.setStyle('fontSize', 24);

        // Apply the new style using a class selector.
        // This maintains the values of the existing style
        // properties that are not over-ridden by the new
        // CSSStyleDeclaration.
        StyleManager.setStyleDeclaration(".myButtonStyle",
            myDynStyle, true);

        // You can also apply the new style by setting the
        // value of the  styleName property in ActionScript.
        myOtherButton.styleName=myDynStyle;

     }
  ]]></mx:Script>

  <mx:Button id="myButton"
        label="Click Me"
        styleName="myButtonStyle"
  />

  <mx:Button id="myOtherButton" label="Click Me"/>

</mx:Application>
```

To remove a CSSStyleDeclaration object, use the StyleManager's `clearStyleDeclaration()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ClearStyleDeclarationExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.styles.StyleManager;

     private var myDynStyle:CSSStyleDeclaration;

     private function initApp():void {
        myDynStyle = new CSSStyleDeclaration('myDynStyle');

        myDynStyle.setStyle('color', 'blue');
        myDynStyle.setStyle('fontFamily', 'georgia');
        myDynStyle.setStyle('themeColor', 'green');
        myDynStyle.setStyle('fontSize', 24);

        StyleManager.setStyleDeclaration(".myButtonStyle",
            myDynStyle, true);

     }

     private function resetStyles():void {
        StyleManager.clearStyleDeclaration(".myButtonStyle",
            true);
     }

  ]]></mx:Script>

  <mx:Button id="myButton"
        label="Click Me"
        styleName="myButtonStyle"
        click="resetStyles()"
  />

</mx:Application>
```

Using the `clearStyleDeclaration()` method removes only the specified selector's styles. If you apply a class selector to a component, and then call the method on that component's class selector, the component's type selector styles remain.

The `setStyleDeclaration()` and `clearStyleDeclaration()` methods are computationally expensive. You can prevent Adobe Flash Player from applying or clearing the new styles immediately by setting the `update` parameter to `false`.

The following example sets new class selectors on different targets, but does not trigger the update until the last style declaration is applied:

```
<?xml version="1.0"?>
<!-- styles/UpdateParameter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.styles.StyleManager;

     private var myButtonStyle:CSSStyleDeclaration =
        new CSSStyleDeclaration('myButtonStyle');
     private var myLabelStyle:CSSStyleDeclaration =
        new CSSStyleDeclaration('myLabelStyle');
     private var myTextAreaStyle:CSSStyleDeclaration =
        new CSSStyleDeclaration('myTextAreaStyle');

     private function initApp():void {
        myButtonStyle.setStyle('color', 'blue');
        myLabelStyle.setStyle('color', 'blue');
        myTextAreaStyle.setStyle('color', 'blue');
     }

     private function applyStyles():void {
        StyleManager.setStyleDeclaration("Button", myButtonStyle, false);
        StyleManager.setStyleDeclaration("Label", myLabelStyle, false);
       StyleManager.setStyleDeclaration("TextArea", myTextAreaStyle, true);
     }
  ]]></mx:Script>

  <mx:Button id="myButton" label="Click Me" click="applyStyles()"/>
  <mx:Label id="myLabel" text="This is a label"/>
  <mx:TextArea id="myTextArea" text="This is a TextArea"/>

</mx:Application>
```

When you pass `false` for the `update` parameter, Flash Player stores the selector but does not apply the style. When you pass `true` for the `update` parameter, Flash Player recomputes the styles for every visual component in the application.

Using the setStyle() and getStyle() methods

You cannot get or set style properties directly on a component as you can with other properties. Instead, you set style properties at run time by using the `getStyle()` and `setStyle()` ActionScript methods. When you use the `getStyle()` and `setStyle()` methods, you can access the style properties of instances of objects or of style sheets.

Every Flex component exposes these methods. When you are instantiating an object and setting the styles for the first time, you should try to apply style sheets rather than use the `setStyle()` method because it is computationally expensive. This method should only be used when you are changing an object's styles during run time. For more information on improving performance with the `setStyle()` method, see "Improving performance with the setStyle() method" on page 742.

Setting styles

The `getStyle()` method has the following signature:

```
return_type componentInstance.getStyle(property_name)
```

The *`return_type`* depends on the style that you access. Styles can be of type String, Number, Boolean, or, in the case of skins, Class. The *`property_name`* is a String that indicates the name of the style property; for example, `fontSize`, or `borderStyle`.

The `setStyle()` method has the following signature:

```
componentInstance.setStyle(property_name, property_value)
```

The *`property_value`* sets the new value of the specified property. To determine valid values for properties, see *Adobe Flex 2 Language Reference*.

The following example uses the `getStyle()` and `setStyle()` methods to change the Button's `fontSize` style and display the new size in the TextInput:

```
<?xml version="1.0"?>
<!-- styles/SetSizeGetSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        fontSize: 10pt;
        color: Blue;
     }
     .myClass {
        fontFamily: Arial, Helvetica, "_sans";
        color: Red;
        fontSize: 22;
        fontWeight: bold;
     }
  </mx:Style>
  <mx:Script><![CDATA[
     public function showStyles():void {
        lb1.text = String(ip1.getStyle("fontSize"));
     }
     public function setNewStyles(newSize:Number):void {
        ip1.setStyle("fontSize",newSize);
     }
  ]]></mx:Script>
  <mx:VBox id="vb">
     <mx:TextInput styleName="myClass" text="My attrs" id="ip1"
        width="400"/>
     <mx:Label id="lb1" text="" width="400"/>
     <mx:Button label="Get Style" click="showStyles();"/>
     <mx:Button label="Set Style" click="setNewStyles(Number(ip2.text));"/>
     <mx:TextInput text="" id="ip2" width="50"/>
  </mx:VBox>
</mx:Application>
```

You can use the `getStyle()` method to access style properties regardless of how they were set. If you defined a style property as a tag property inline rather than in an `<mx:Style>` tag, you can get and set this style. You can override style properties that were applied in any way, such as in an `<mx:Style>` tag or in an external style sheet.

The following example sets a style property inline, and then reads that property with the `getStyle()` method:

```
<?xml version="1.0"?>
<!-- styles/GetStyleInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     private function readStyle():void {
        myLabel.text = "Style: " + myLabel.getStyle("fontStyle");
     }
  ]]></mx:Script>
  <mx:VBox width="500"
     height="200">
     <mx:Button id="b1" click="readStyle()" label="Get Style"/>
     <mx:Label fontStyle="italic" id="myLabel"/>
  </mx:VBox>
</mx:Application>
```

When setting color style properties with the `setStyle()` method, you can use the hexadecimal format or the VGA color name, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatStyleManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     public function initApp():void {
        StyleManager.getStyleDeclaration("Button").
            setStyle("themeColor",0x6666CC);
        StyleManager.getStyleDeclaration("Button").
            setStyle("color","Blue");
     }

     public function changeStyles(e:Event):void {
        e.currentTarget.setStyle("themeColor", 0xFF0099);
        e.currentTarget.setStyle("color", "Red");
     }
  ]]></mx:Script>
  <mx:Button id="myButton"
        label="Click Here"
        click="changeStyles(event)"
  />
</mx:Application>
```

When you get a `color` style property with the `getStyle()` method, Flex returns an integer that represents the hexadecimal value of the style property. To convert this to its hexadecimal format, you use the color variable's `toString()` method and pass it the value 16 for the radix (or base):

```
<?xml version="1.0"?>
<!-- styles/ColorFormatNumericValue.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">

  <mx:Style>
     Button {
        color: #66CCFF;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     [Bindable]
     private var n:Number;

     private function initApp():void {
        n = myButton.getStyle("color");
     }

     public function changeStyles(e:Event):void {
        if (myButton.getStyle("color").toString(16) == "ff0000") {
           myButton.setStyle("color", 0x66CCFF);
        } else {
           myButton.setStyle("color", "Red");
        }

        n = myButton.getStyle("color"); //// Returns 16711680
     }
  ]]></mx:Script>

  <mx:Button id="myButton" label="Click Here" click="changeStyles(event)"/>

  <mx:Label id="myLabel" text="0x{n.toString(16).toUpperCase()}"/>

</mx:Application>
```

When you use the `setStyle()` method to change an existing style (for example, to set the `color` property of a Button control to something other than 0x000000, the default), Flex does not overwrite the original style setting. You can return to the original setting by setting the style property to `null`. The following example toggles the color of the Button control between blue and the default by using this technique:

```
<?xml version="1.0"?>
<!-- styles/ResetStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
        public function toggleStyle():void {
            if (cb1.selected == true) {
                b1.setStyle("color","blue");
                b1.setStyle("fontSize", 8);
            } else {
                b1.setStyle("color", null);
                b1.setStyle("fontSize", null);
            }
        }
        ]]>
    </mx:Script>

    <mx:Style>

        Button {
            color: red;
            fontSize: 25;
        }

    </mx:Style>

    <mx:Button id="b1" label="Click Me"/>

    <mx:CheckBox id="cb1"
        label="Set Style/Unset Style"
        click="toggleStyle()"
        selected="false"
    />
</mx:Application>
```

Improving performance with the setStyle() method

Run-time cascading styles are very powerful, but you should use them sparingly and in the correct context. Dynamically setting styles on an instance of an object means accessing the UIComponent's `setStyle()` method. The `setStyle()` method is one of the most resource-intensive calls in the Flex framework because the call requires notifying all the children of the newly styled object to do another style lookup. The resulting tree of children that must be notified can be quite large.

A common mistake that impacts performance is overusing or unnecessarily using the `setStyle()` method. In general, you only need the `setStyle()` method when you want to change styles on existing objects. Do not use it when setting up styles for an object for the first time. Instead, set styles in an `<mx:Style>` block, through an external CSS style sheet, or as global styles. It is important to initialize your objects with the correct style information, if you do not expect these styles to change while your program executes (whether it is your application, a new view in a navigator container, or a dynamically created component).

Some applications must call the `setStyle()` method during the application or object instantiation. If this is the case, call the `setStyle()` method early in the instantiation phase. Early in the instantiation phase means setting styles from the component or application's `preinitialize` event, instead of the `creationComplete` or other event. By setting the styles as early as possible during initialization, you avoid unnecessary style notification and lookup. For more information about the component startup life cycle, see Chapter 6, "Improving Startup Performance," in *Building and Deploying Flex 2 Applications*.

Using inline styles

You can set style properties as properties of the component in the MXML tag. Inline style definitions take precedence over any other style definitions. The following example defines a type selector for Button components, but then overrides the `color` with an inline definition:

```
<?xml version="1.0"?>
<!-- styles/InlineOverride.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        fontSize: 10pt;
        fontStyle: italic;
        color: #FF0000;
     }
  </mx:Style>
  <mx:Button label="Button Type Selector Color"/>
  <mx:Button color="0x999942" label="Inline Color"/>
</mx:Application>
```

When setting style properties inline, you must adhere to the ActionScript style property naming syntax rather than the CSS naming syntax. For example, you can set a Button control's `fontSize` property as `font-size` or `fontSize` in an `<mx:Style>` declaration, but you must set it as `fontSize` in a tag definition:

```
<?xml version="1.0"?>
<!-- styles/CamelCase.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myFontStyle {
        fontSize: 15;
     }
     .myOtherFontStyle {
        font-size: 15;
     }
  </mx:Style>
  <mx:Button id="myButton" styleName="myFontStyle" label="Click Here"/>
  <mx:Button id="myButton2" styleName="myOtherFontStyle" label="Click
Here"/>
  <mx:Button id="myButton3" fontSize="15" label="Click Here"/>
</mx:Application>
```

For more information, see "About property and selector names" on page 712.

When setting color style properties inline, you can use the hexadecimal format or the VGA color name, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/ColorFormatInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Button id="myButton" themeColor="0x6666CC" color="Blue" label="Click
Here"/>

</mx:Application>
```

You can remove an inline style definition by using the `clearStyle()` method.

You can bind inline style properties to variables, as long as you tag the variable as `[Bindable]`. The following example binds the value of the `backgroundColor` property of the HBox controls to the value of the `colorValue` variable:

```
<?xml version="1.0"?>
<!-- styles/PropertyBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     [Bindable]
     public var colorValue:int = 0x333999;

     public function changeHBoxStyle():void {
        colorValue = cp.selectedColor;
     }
  ]]></mx:Script>
  <mx:HBox width="100" height="100" backgroundColor="{colorValue}"/>
  <mx:ColorPicker id="cp" showTextField="true" change="changeHBoxStyle()"
selectedColor="0x333999"/>
</mx:Application>
```

Binding a style property can be a computationally expensive operation. You should only use this method of applying style properties when absolutely necessary.

Loading style sheets at run time

You can load style sheets at run time by using the StyleManager. These style sheets take the form of SWF files that are dynamically loaded while your Flex application runs.

By loading style sheets at run time, you can load images (for graphical skins), fonts, type and class selectors, and programmatic skins into your Flex application without embedding them at compile time. This lets skins and fonts be partitioned into separate SWF files, away from the main application. As a result, the application's SWF file size is smaller, which reduces the initial download time. However, the first time a run-time style sheet is used, it takes longer for the styles and skins to be applied because Flex must download the necessary CSS-based SWF file.

Loading style sheets at run time is a three-step process:

1. Write a CSS file for your application.
2. Compile the CSS file in a SWF file by using the mxmlc compiler.
3. Call the `StyleManager.loadStyleDeclarations()` method in your Flex application. This method loads the CSS-based SWF file into your application. When this method executes, Flex loads the new CSSStyleDeclarations into the StyleManager.

You can load multiple style sheets that define the same styles. After you set a style, subsequent style sheets can overwrite previous ones if they have common selectors. Styles loaded with run-time style sheets do not completely replace compile-time styles, however. They just override them until the run-time style sheets are unloaded. At that point, Flex reverts to the compile-time style settings. Compile-time style settings include any default styles sheets that were loaded at compile time, theme files loaded by using the `theme` compiler option, and styles set by using the `<mx:Style>` block inside an MXML file.

Creating a run-time style sheet

To load style sheets at run time, you must first create a style sheet that is compiled into a SWF file. A run-time style sheet is like any other style sheet. It can be a simple style sheet that sets basic style properties, as the following example shows:

```
/* styles/runtime/assets/BasicStyles.css */
Button {
    fontSize:    24;
    color: #FF9933;
}

Label {
    fontSize:    24;
    color: #FF9933;
}
```

Or the style sheet can be a complex style sheet that embeds programmatic and graphical skins, fonts, and other style properties, and uses type and class selectors, as the following example shows:

```
/* styles/runtime/assets/ComplexStyles.css */
Application {
    backgroundImage : "assets/greenBackground.gif";
    theme-color: #9DBAEB;
}
Button {
    fontFamily: Tahoma;
    color: #000000;
    fontSize: 11;
    fontWeight: normal;
    text-roll-over-color: #000000;
    upSkin: Embed(source="../../../assets/orb_up_skin.gif");
    overSkin: Embed(source="../../../assets/orb_over_skin.gif");
    downSkin: Embed(source="../../../assets/orb_down_skin.gif");
}

.noMargins {
    margin-right: 0;
    margin-left: 0;
    margin-top: 0;
    margin-bottom: 0;
    horizontal-gap: 0;
    vertical-gap: 0;
}
```

You cannot load an uncompiled CSS file into your Flex application at run time. You must compile it into a SWF file before loading it.

Compiling the CSS-based SWF file

Before you can load a style sheet at run time, you must compile the style sheet into a SWF file.

To compile the CSS file into a SWF file, you use the mxmlc command-line compiler. The default result of the compilation is a SWF file with the same name as the CSS file, but with the .swf extension. The following example produces the BasicStyles.swf file:

```
> mxmlc /skins/runtime/BasicStyles.css
```

The style sheet that you compile into a SWF file must use a .css filename extension.

When you compile your Flex application, the compiler does not perform any compile-time link checking against the CSS-based SWF files used by the application. This means that you are not required to create the SWF file before you compile your main application.

Loading style sheets at run time

You load a CSS-based SWF file at run time by using the StyleManager's `loadStyleDeclarations()` method. To use this method, you must import the mx.core.StyleManager class.

The following example loads a style sheet when you click the button:

```
<?xml version="1.0"?>
<!-- styles/runtime/BasicApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
        import mx.styles.StyleManager;

        public function applyRuntimeStyleSheet():void {
            StyleManager.loadStyleDeclarations("assets/BasicStyles.swf")
        }
        ]]>
    </mx:Script>

    <mx:Label text="Click the button to load a new CSS-based SWF file"/>
    <mx:Button id="b1" label="Click Me" click="applyRuntimeStyleSheet()"/>

</mx:Application>
```

The first parameter of the `loadStyleDeclarations()` method is the location of the style sheet to load. The location can be local or remote.

The following example shows loading a local and a remote SWF file:

```
// Load a locally-accessible style sheet:
StyleManager.loadStyleDeclarations("assets/LocalStyles.swf");
// Load a remote style sheet:
StyleManager.loadStyleDeclarations("http://www.domain2.com/styles/
  RemoteStyles.swf", true, true);
```

If the style sheet is local (from the same domain as the loading application), you can load it without any additional settings. If the location of the style sheet is remote (in any domain that is not an exact match of the loading application's domain), you must set the third parameter of the `loadStyleDeclarations()` method, `trustContent`, to `true`. Loading remote style sheets does not require a crossdomain.xml file that gives the loading application permission to load the SWF file.

Also, to use remote style sheets, you must compile the loading application with network access (have the `use-network` compiler property set to `true`, the default). If you compile and run the application on a local file system, you might not be able to load a remotely accessible SWF file.

The `loadStyleDeclarations()` method returns an instance of the IEventDispatcher class. You can use this object to trigger events based on the success of the style sheet's loading. You have access to the `StyleEvent.PROGRESS`, `StyleEvent.COMPLETE`, and `StyleEvent.ERROR` events of the loading process.

The following application calls a method when the style sheet finishes loading:

```
<?xml version="1.0"?>
<!-- styles/runtime/StylesEventApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">

    <mx:Script>
        <![CDATA[
        import mx.styles.StyleManager;
        import mx.events.StyleEvent;

        public function init():void {
            var myEvent:IEventDispatcher =
                StyleManager.loadStyleDeclarations("assets/ACBStyles.swf");
            myEvent.addEventListener(StyleEvent.COMPLETE, getImage);
        }

        private function getImage(event:StyleEvent):void {
            map1.source = acb.getStyle("dottedMap");
        }
        ]]>
    </mx:Script>

    <mx:ApplicationControlBar id="acb" width="100%" styleName="homeMap">
        <mx:Image id="map1"/>
        <mx:Button label="Submit"/>
    </mx:ApplicationControlBar>

</mx:Application>
```

The second parameter of the `loadStyleDeclarations()` method is `update`. Set the `update` parameter to `true` to force an immediate update of the styles. Set it to `false` to avoid an immediate update of the styles in the application. The styles are updated the next time you call this method or the `unloadStyleDeclarations()` method with the `update` property set to `true`.

Each time you call the `loadStyleDeclarations()` method with the `update` parameter set to `true`, Adobe Flash Player reapplies all styles to the display list, which can degrade performance. If you load multiple CSS-based SWF files at the same time, you should set the `update` parameter to `false` for all but the last call to this method. As a result, Flash Player only applies the styles once for all new style SWF files rather than once for each new style SWF.

The following example loads three style SWF files, but does not apply them until the third one is loaded:

```
<?xml version="1.0"?>
<!-- styles/runtime/DelayUpdates.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">

    <mx:Script>
        <![CDATA[
        import mx.styles.StyleManager;
        import mx.events.StyleEvent;

        public function init():void {
            StyleManager.loadStyleDeclarations(
                "assets/ButtonStyles.swf", false);
            var myEvent:IEventDispatcher =
                StyleManager.loadStyleDeclarations(
                "assets/LabelStyles.swf", false);
            myEvent.addEventListener(StyleEvent.COMPLETE, doUpdate);
        }

        public function doUpdate(event:StyleEvent):void {
            StyleManager.loadStyleDeclarations(
                "assets/ACBStyles.swf", true);
        }
        ]]>
    </mx:Script>

    <mx:Label text="This is a label"/>

    <mx:ApplicationControlBar id="acb" width="100%">
        <mx:Button label="Submit"/>
    </mx:ApplicationControlBar>

</mx:Application>
```

Unloading style sheets at run time

You can unload a style sheet that you loaded at run time. You do this by using the StyleManager's `unloadStyleDeclarations()` method. The result of this method is that all style properties set by the specified style SWF files are returned to their defaults.

The following example loads and unloads a style SWF when you toggle the check box:

```
<?xml version="1.0"?>
<!-- styles/runtime/UnloadStyleSheets.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
        import mx.styles.StyleManager;

        public function toggleStyleSheet():void {
            if (cb1.selected == true) {
                StyleManager.loadStyleDeclarations(
                    "assets/ButtonStyles.swf", true, false)
                StyleManager.loadStyleDeclarations(
                    "assets/LabelStyles.swf", true, false)
            } else {
                StyleManager.unloadStyleDeclarations(
                    "assets/ButtonStyles.swf", true);
                StyleManager.unloadStyleDeclarations(
                    "assets/LabelStyles.swf", true);
            }
        }
        ]]>
    </mx:Script>

    <mx:Button id="b1" label="Click Me"/>

    <mx:Label id="l1" text="Click the button"/>

    <mx:CheckBox id="cb1"
        label="Load style sheet"
        click="toggleStyleSheet()"
        selected="false"
    />
</mx:Application>
```

Using run-time style sheets in custom components

You can use run-time style sheets in custom components. To do this, you generally call the `loadStyleDeclaration()` method after the component is initialized. If the style sheet contains class selectors, you then apply them by setting the `styleName` property.

The following example defines style properties and skins in a class selector named specialStyle:

```
/* styles/runtime/assets/CustomComponentStyles.css */
.specialStyle {
    fontSize:    24;
    color: #FF9933;
    upSkin: Embed(source="SubmitButtonSkins.swf", symbol="MyUpSkin");
    overSkin: Embed(source="SubmitButtonSkins.swf", symbol="MyOverSkin");
    downSkin: Embed(source="SubmitButtonSkins.swf", symbol="MyDownSkin");
}
```

The following example custom component loads this style sheet and applies the specialStyle class selector to itself during initialization:

```
// styles/runtime/MyButton.as -->
package {

    import mx.controls.Button;
    import mx.events.*;
    import mx.styles.StyleManager;

    public class MyButton extends Button {

        public function MyButton() {
            addEventListener(FlexEvent.INITIALIZE,
                initializeHandler);
        }

         // Gets called when the component has been initialized
        private function initializeHandler(event:FlexEvent):void {
            StyleManager.loadStyleDeclarations(
                "assets/CustomComponentStyles.swf");
            this.styleName = "specialStyle";
        }
    }
}
```

Using theme SWC files as run-time style sheets

If you have an existing theme SWC file, you can use it as a run-time style sheet. To do this, you must extract the CSS file from the theme SWC file. You then compile the style SWF file by passing the remaining SWC file as a library.

The following steps show this process using the command line:

1. Extract the CSS file from the SWC file using PKZip or similar archiving utility; for example:

   ```
   $ unzip haloclassic.swc defaults.css
   ```

2. (Optional) Rename the CSS file to a meaningful name. This is the name of your style SWF file. The following example renames the defaults.css file to haloclassic.css:

```
$ mv defaults.css haloclassic.css
```

3. Compile the style SWF file. Add the theme SWC file to the style SWF file by using the `include-libraries` option, as the following example shows:

```
$ mxmlc -include-libraries=haloclassic.swc haloclassic.css
```

If you have multiple CSS files inside a theme SWC file, you must extract all of them before compiling the style SWF file.

Using filters in Flex

You can use Adobe Flash filters to apply style-like effects to Flex components, such as Labels and Text. You can apply filters to any visual Flex component that is derived from UIComponent. Filters are not styles because you cannot apply them with a style sheet or the `setStyle()` method. The result of a filter, though, is often thought of as a style.

Filters are in the flash.filters.* package, and include the DropShadowFilter, GlowFilter, and BlurFilter classes. To apply a filter to a component with MXML, you add the flash.filters package to the local namespace. You then set the value of the Flex component's `filters` property to the filter class. The `filters` property is a property inherited from the DisplayObject class.

The following example applies a drop shadow to a Label control by using expanded MXML syntax and inline syntax:

```
<?xml version="1.0"?>
<!-- styles/ApplyFilterInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:flash="flash.filters.*">
  <!-- Apply filter using MXML syntax to set properties. -->
  <mx:Label text="DropShadowFilter" fontSize="20">
    <mx:filters>
     <flash:DropShadowFilter distance="10" angle="45"/>
    </mx:filters>
  </mx:Label>

  <!-- Apply filter and set properties inline. -->
  <mx:Label text="DropShadowFilter (inline)" fontSize="20"
     filters="{[new DropShadowFilter(10, 45)]}"/>
</mx:Application>
```

You can apply filters in ActionScript. You do this by importing the flash.filters.* package, and then adding the new filter to the filters Array of the Flex control. The following example applies a white shadow to the Label control when the user clicks the button:

```
<?xml version="1.0"?>
<!-- styles/ApplyFilterAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
     import flash.filters.*;

     public function addFilter():void {
        // First four properties are distance, angle, color, and alpha.
        var f:DropShadowFilter = new DropShadowFilter(5,30,0xFFFFFF,.8);
        var myFilters:Array = new Array();
        myFilters.push(f);
        label1.filters = myFilters;
     }
  </mx:Script>
  <mx:Label id="label1" text="ActionScript-applied filter"/>
  <mx:Button id="b1" label="Add Filter" click="addFilter()"/>
</mx:Application>
```

You cannot bind the filter properties to other values.

If you change a filter, you must reassign it to the component so that the changes take effect. The following example changes the color of the filters when you click the button:

```
<?xml version="1.0"?>
<!-- styles/FilterChange.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="createFilters()">
  <mx:Script><![CDATA[
     import flash.filters.*;

     private var myBlurFilter:BlurFilter;
     private var myGlowFilter:GlowFilter;
     private var myBevelFilter:BevelFilter;
     private var myDropShadowFilter:DropShadowFilter;

     private var color:Number = 0xFF33FF;

     public function createFilters():void {

        myBlurFilter = new BlurFilter(4, 4, 1);

        myGlowFilter = new GlowFilter(color, .8, 6, 6, 2, 1,
            false, false);

        myDropShadowFilter = new DropShadowFilter(15, 45,
            color, 0.8, 8, 8, 0.65, 1, false, false);

        myBevelFilter = new BevelFilter(5, 45, color, 0.8,
            0x333333, 0.8, 5, 5, 1, BitmapFilterQuality.HIGH,
            BitmapFilterType.INNER, false);

        applyFilters();
     }

     public function applyFilters():void {
        rte1.filters = [myGlowFilter];
        b1.filters = [myDropShadowFilter];
        dc1.filters = [myBevelFilter];
        hs1.filters = [myBlurFilter];
     }

     public function changeFilters():void {
        color = 0x336633;
        createFilters();
     }
  ]]></mx:Script>

  <mx:RichTextEditor id="rte1"/>

  <mx:DateChooser id="dc1"/>
```

```
    <mx:HSlider id="hs1"/>

    <mx:Button id="b1" label="Click me" click="changeFilters()"/>

</mx:Application>
```

About themes

A *theme* defines the look and feel of a Flex application. A theme can define something as simple as the color scheme or common font for an application, or it can be a complete reskinning of all the components used by the application.

Themes usually take the form of a SWC file. However, themes can also be a CSS file and embedded graphical resources, such as symbols from a SWF file. Theme SWC files can also be compiled into style SWF files so that they can be loaded at run time. For more information, see "Using theme SWC files as run-time style sheets" on page 752.

To apply the contents of that SWC file to your Flex application, use the instructions in "Using themes" on page 756. To create your own theme, use the instructions in "Creating a theme SWC file" on page 758. You can affect the theme of a Flex application without creating a new theme. You do this with the `themeColor` property. For more information, see "Creating themes" on page 849.

The default theme, Halo, is a combination of graphical and programmatic skins in a SWC file. It is defined by the defaults.css file in the framework.swc file. This file sets many properties of the Flex components. In some cases it uses classes in the mx.skins.halo package. Flex also includes several predefined themes that you can apply to your applications. For more information, see "About the included theme files" on page 758.

Using themes

Themes generally take the form of a theme SWC file. These SWC files contain style sheets and skinning assets. You use the assets inside theme SWC files for programmatic skins or graphical assets, such as SWF, GIF, or JPEG files. Themes can also contain just stand-alone CSS files.

Packaging a theme as a SWC file rather than as a loose collection of files has the following benefits:

- SWC files are easier to distribute.
- SWC files cannot be modified and reapplied to the application without recompiling.

- SWC files are precompiled. This reduces application compile time, compared to compile time for skin classes that are not in a SWC file.

You apply a theme to your Flex application by specifying the SWC or CSS file with the `theme` compiler option. The following example uses the mxmlc command-line compiler to compile an application that uses the BullyBuster theme SWC file:

```
mxmlc -theme c:/theme/BullyBuster.swc c:/myfiles/flex2/misc/MainApp.mxml
```

When compiling an application by using options in the flex-config.xml file, you specify the theme as follows:

```
<compiler>
  <theme>
    <filename>c:/theme/BullyBuster.swc</filename>
  </theme>
</compiler>
```

When you add a SWC file to the list of themes, the compiler adds the classes in the SWC file to the application's `library-path`, and applies any CSS files contained in the SWC file to your application. The converse is not true, however. If you add a SWC file to the `library-path`, but do not specify that SWC file with the `theme` option, the compiler does not apply CSS files in that SWC file to the application.

For more information on using Flex compilers, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

You can specify more than one theme file to be applied to the application. If there are no overlapping styles, both themes are applied completely. The ordering of the theme files is important, though. If you specify the same style property in more than one theme file, Flex uses the property from the last theme in the list. In the following example, style properties in the Wooden.css file take precedence, but unique properties in the first two theme files are also applied:

```
<compiler>
  <theme>
    <filename>../themes/Institutional.css</filename>
    <filename>../themes/Ice.css</filename>
    <filename>../themes/Wooden.css</filename>
  </theme>
</compiler>
```

About the included theme files

Flex includes several themes that you can use without making any changes. With the exception of the default Halo theme, all of these themes are located in the *flex_install_dir*/frameworks/themes directory for the SDK. For Flex Data Services, the themes are located in the *flex_install_dir*/flex_sdk_2/frameworks/themes directory. For Flex Builder, the themes are located in the *flex_install_dir*/Flex SDK 2/frameworks/themes directory.

The following table describes these themes:

Theme	Description
Aeon Graphical	Contains the AeonGraphical.css file and the AeonGraphical.swf file. This is the graphical version of the default Halo theme, which is defined by the defaults.css file and graphical assets inside the framework.swc file. The source FLA file for the graphical assets is in the AeonGraphical Source directory. You change the graphical assets in this file by using the Flash IDE. You must ensure that you export the SWF file, which is used by the theme in Flex. This theme is provided as a starting point for graphically reskinning your application. While it might be more efficient, you lose some ability to apply run-time styles if you use it.
Halo	Provides the default Flex look and feel. The Halo theme uses the defaults.css file in the framework.swc file and the classes in the mx.skins.halo package. This is the default theme. For more information about the default theme assets, see "About the default style sheet" on page 727.
HaloClassic	Provides the look and feel of previous versions of Flex. This theme comprises the haloclassic.swc file. Included in this file are the CSS file and SWF file that provide graphical assets.
Ice	Contains the Ice.css file.
Institutional	Contains the Institutional.css file.
Smoke	Contains the Smoke.css file and a background JPEG file.
Wooden	Contains the Wooden.css file and a background JPEG file.

Creating a theme SWC file

To build a theme SWC file, you create a CSS file, and include that file plus the graphical and programmatic assets in the SWC file. To do this, you use the `include-file` option of the compc utility.

A good starting point to creating a new theme is to customize the defaults.css file in the framework.swc file. The following sections describe these steps in detail.

Creating a theme style sheet

A theme typically consists of a style sheet and any assets that the style sheet uses. These assets can be graphic files or programmatic skin classes.

To add graphic files to the theme's CSS file, you use the `Embed` statement. The following example defines new graphical skins for the Button class:

```
Button {
  upSkin: Embed("upIcon.jpg");
  downSkin: Embed("downIcon.jpg");
  overSkin: Embed("overIcon.jpg");
}
```

The name you provide for the `Embed` keyword is the same name that you use for the skin asset when compiling the SWC file.

To add programmatic skin classes to the theme's CSS file, you use the `ClassReference` statement to specify the class name (not the file name), as the following example shows:

```
Button {
  upSkin:ClassReference('myskins.ButtonSkin');
  downSkin:ClassReference('myskins.ButtonSkin');
  overSkin:ClassReference('myskins.ButtonSkin');
}
```

In the preceding example, all skins are defined by a single programmatic skin class file. For more information on creating programmatic skins, see "Programmatic skinning" on page 816.

The CSS file can include any number of class selectors, as well as type selectors. Style definitions for your theme do not have to use skins. You can simply set style properties in the style sheet, as the following example shows:

```
ControlBar {
  borderStyle: "controlBar";
  paddingBottom: 10;
  paddingLeft: 10;
  paddingRight: 10;
  paddingTop: 10;
  verticalAlign: "middle";
}
```

This type selector for ControlBar components is taken from the default style sheet, defaults.css, inside the framework.swc file. It can be a good place to start when designing a theme. For more information about defaults.css, see "About the default style sheet" on page 727.

Compiling a theme SWC file

You compile a theme SWC file by using the `include-file` and `include-classes` options of the Flex component compiler. This is the same compiler that creates component libraries and Runtime Shared Libraries (RSLs).

You invoke the component compiler either with the compc command line utility or when creating a Library Project in Adobe Flex Builder.

You use the `include-file` option to add the CSS file and graphics files to the theme SWC file. This is described in "Using the include-file option to compile theme SWC files" on page 760. You use the `include-classes` option to add programmatic skin classes to the theme SWC file. This is described in "Using the include-classes option to compile theme SWC files" on page 761.

To simplify the commands for compiling theme SWC files, you can use configuration files. For more information, see "Using a configuration file to compile theme SWC files" on page 761.

Using the include-file option to compile theme SWC files

The `include-file` option takes two arguments: a name and a path. The name is the name that you use to refer to that asset as in the CSS file. The path is the file system path to the asset. When using `include-file`, you are not required to add the resources to the source path. The following command line example includes the upIcon.jpg asset in the theme SWC file:

```
-include-file upIcon.jpg c:/myfiles/themes/assets/upIcon.jpg
```

You also specify the CSS file as a resource to include in the theme SWC file. You must include the .css extension when you provide the name of the CSS file, otherwise Flex does not recognize it as a style sheet and does not apply it to your application.

You use the component compiler to compile theme assets, such as a style sheet and graphical skins, into a SWC file. The following example compiles a theme by using the compc command-line compiler:

```
compc -include-file mycss.css c:/myfiles/themes/mycss.css
  -include-file upIcon.jpg c:/myfiles/themes/assets/upIcon.jpg
  -include-file downIcon.jpg c:/myfiles/themes/assets/downIcon.jpg
  -include-file overIcon.jpg c:/myfiles/themes/assets/overIcon.jpg
  -o c:/myfiles/themes/MyTheme.swc
```

You cannot pass a list of assets to the `include-file` option. Each pair of arguments must be preceded by `-include-file`. Most themes use many skin files and style sheets, which can be burdensome to enter each time you compile. To simplify this, you can use the `load-config` option to specify a file that contains configuration options, such as multiple `include-file` options. For more information, see "Using a configuration file to compile theme SWC files" on page 761.

Using the include-classes option to compile theme SWC files

The `include-classes` option takes a single argument: the name of the class to include in the SWC file. You pass the class name and not the class file name (for example, MyButtonSkin rather than MyButtonSkin.as). The class must be in your source path when you compile the SWC file.

The following command-line example compiles a theme SWC file that includes the CSS file and a single programmatic skin class, MyButtonSkin, which is in the themes directory:

```
compc -source-path c:/myfiles/flex2/themes
  -include-file mycss.css c:/myfiles/flex2/themes/mycss.css
  -include-classes MyButtonSkin -o c:/myfiles/flex2/themes/MyTheme.swc
```

You can pass a list of classes to the `include-classes` option by space-delimiting each class, as the following example shows:

```
-include-classes MyButtonSkin MyControlBarSkin MyAccordionHeaderSkin
```

For more information about creating skin classes, see Chapter 20, "Using Skins," on page 805.

Using a configuration file to compile theme SWC files

Using a configuration file is generally more verbose than passing options on the command line, but it can make the component compiler options easier to read and maintain. For example, you could replace a command line with the following entries in a configuration file:

```
<?xml version="1.0"?>
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
  <output>MyTheme.swc</output>
  <include-file>
    <name>mycss.css</name>
    <path>c:/myfiles/themes/mycss.css</path>
  </include-file>
  <include-file>
    <name>upIcon.jpg</name>
    <path>c:/myfiles/themes/assets/upIcon.jpg</path>
  </include-file>
  <include-file>
    <name>downIcon.jpg</name>
```

```
    <path>c:/myfiles/themes/assets/downIcon.jpg</path>
  </include-file>
  <include-file>
    <name>overIcon.jpg</name>
    <path>c:/myfiles/themes/assets/overIcon.jpg</path>
  </include-file>
  <include-classes>
    <class>MyButtonSkin</class>
    <class>MyAccordionHeaderSkin</class>
    <class>MyControlBarSkin</class>
  </include-classes>
</flex-config>
```

You can use the configuration file with compc by using the `load-config` option, as the following example shows:

```
compc -load-config myconfig.xml
```

You can also pass a configuration file to the Flex Builder component compiler. For more information on using the component compilers, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

CHAPTER 19

Using Fonts

19

You can include fonts in your Adobe Flex applications. Although it is easier to use the default device fonts, you can embed other fonts so that you can apply special effects to text-based controls, such as rotating and fading. This topic describes the differences between device fonts and embedded fonts. It describes how to efficiently embed double-byte fonts. It also shows multiple ways that you can embed fonts in your Adobe Flex applications, from using Cascading Style Sheets (CSS) syntax to using fonts embedded inside external SWF files.

Contents

About fonts

When you compile a Flex application, the application stores the names of the fonts that you used to create the text. Adobe Flash Player 9 uses the font names to locate identical or similar fonts on the user's system when the Flex application runs. You can also embed fonts in the Flex application so that the exact font is used, regardless of whether the client's system has that font.

You define the font that appears in each of your components by using the `fontFamily` style property. You can set this property in an external style sheet, a `<mx:Style>` block, or inline. This property can take a list of fonts, as the following example shows:

```
.myClass {
  fontFamily: Arial, Helvetica;
  color: Red;
  fontSize: 22;
  fontWeight: bold;
}
```

If the client's system does not have the first font in the list, Flash Player attempts to find the second, and so on, until it finds a font that matches. If no fonts match, Flash Player makes a best guess to determine which font the client uses.

Fonts are inheritable style properties. So, if you set a font style on a container, all controls inside that container inherit that style, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/InheritableExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     VBox {
        fontFamily: Helvetica;
        fontSize: 18pt;
     }
     HBox {
        fontFamily: Times;
        fontSize: 18pt;
     }
  </mx:Style>

  <mx:Panel title="Styles Inherited from VBox Type Selector">
     <mx:VBox>
        <mx:Button label="This Button uses Helvetica"/>
        <mx:Label text="This label is in Helvetica"/>
        <mx:TextArea width="400" height="75" text="The text in this
            control uses the Helvetica font because it is inherited from
            the VBox style."
        />
     </mx:VBox>
  </mx:Panel>
  <mx:Panel title="Styles Inherited from HBox Type Selector">
     <mx:HBox>
        <mx:Button label="This Button uses Helvetica"/>
        <mx:Label text="This label is in Helvetica"/>
        <mx:TextArea width="400" height="75"
            text="The text in this control uses the Times font
            because it is inherited from the HBox style."
        />
     </mx:HBox>
  </mx:Panel>
</mx:Application>
```

This example defines the HBox and VBox type selectors' `fontSize` and `fontFamily` properties. Flex applies these styles to all components in the container that support those properties; in these cases, the Button, Label, and TextField controls.

Using device fonts

You can specify any font for the `fontFamily` property. However, not all systems have all font faces, which can result in an unexpected appearance of your controls. The safest course when specifying font faces is to include a device font as a default at the end of the font list. *Device fonts* do not export font outline information and are not embedded in the SWF file. Instead, Flash Player uses whatever font on the local computer most closely resembles the device font.

The following example specifies the device font `_sans` to use if Flash Player cannot find either of the other fonts on the client machine:

```
<?xml version="1.0"?>
<!-- fonts/DeviceFont.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myClass {
        fontFamily: Arial, Helvetica, "_sans";
        color: Red;
        fontSize: 22;
        fontWeight: bold;
     }
  </mx:Style>

  <mx:Panel title="myClass Class Selector with Device Font">
     <mx:VBox styleName="myClass">
        <mx:Button label="Click Me"/>
        <mx:Label text="Label"/>
        <mx:TextArea width="400" height="75" text="The text
            uses the myClass class selector."
        />
     </mx:VBox>
  </mx:Panel>
</mx:Application>
```

NOTE	You must surround device font names with quotation marks when defining them with style declarations.

Flash Player supports three device fonts. The following table describes these fonts:

Font name	Description
`_sans`	The `_sans` device font is a sans-serif typeface; for example, Helvetica or Arial.
`_serif`	The `_serif` device font is a serif typeface; for example, Times Roman.
`_typewriter`	The `_typewriter` device font is a monospace font; for example, Courier.

Using device fonts does not impact the size of the SWF file because the fonts reside on the client. However, using device fonts can impact performance of the application because it requires that Flash Player interact with the local operating system. Also, if you use only device fonts, your selection is limited to three fonts.

Using embedded fonts

Rather than rely on a client machine to have the fonts you specify, you can embed TrueType font families in your Flex application. This means that the font is always available to Flash Player when running the application, and you do not have to consider the implications of a missing font.

Embedded fonts have the following benefits:

- Client environment does not need the font to be installed.
- Embedded fonts are anti-aliased, which means that their edges are smoothed for easier readability. This is especially apparent when the text size is large.
- Embedded fonts can be partially or fully transparent.
- Embedded fonts can be rotated.
- Embedded fonts provide smoother playback when zooming.
- Text appears exactly as you expect when using embedded fonts.
- When you embed a font, you can use the FlashType hinting information that provides clear, high-quality text rendering in SWF files. Using FlashType hinting greatly improves the readability of text, particularly when it is rendered at smaller font sizes. For more information about FlashType hinting, see "Using FlashType hinting" on page 775.

Using embedded fonts is not always the best solution, however. Embedded fonts have the following limitations and drawbacks:

- Embed only TrueType fonts. To embed other font types such as Type 1 PostScript fonts, embed that font in a SWF file that you create in Flash 8, and then embed that SWF file in your Flex application. For more information, see "Embedding fonts from SWF files" on page 791.
- Embedded fonts increase the file size of your application, because the document must contain font outlines for the text. This can result in longer download times for your users.
- Embedded fonts, in general, decrease the legibility of the text at sizes smaller than 10 points. All embedded fonts use anti-aliasing to render the font information on the client screen. As a result, fonts may look fuzzy or illegible at small sizes. To avoid this fuzziness, you can use FlashType hinting to render fonts in Flex applications. For more information, see "Using FlashType hinting" on page 775.

You typically use CSS syntax for embedding fonts in Flex applications. You use the `@font-face` "at-rule" declaration to specify the source of the embedded font, and then define the name of the font by using the `fontFamily` property. You must specify the `@font-face` declaration for each face of the font for the same family that you use. The source can be a local JRE font or one that is accessible by using a file path. You use this font family name in your MXML code to refer to the embedded font.

NOTE	Check your font licenses before embedding any font files in your Flex applications. Fonts might have licensing restrictions that preclude them from being stored as vector information.

If you attempt to embed a font that the Flex compiler cannot find, Flex throws an error and your application does not compile.

Embedded font syntax

To embed TrueType fonts, you use the following syntax in your style sheet or `<mx:Style>` tag:

```
@font-face {
  src:url("location") | local("name");
  fontFamily: alias;
  [fontStyle: italic | oblique | normal;]
  [fontWeight: bold | heavy | normal;]
  [flashType: true | false;]
}
```

The `src` property specifies how the compiler should look for the font, and the name of the font to look for. You can either load a font by its filename (by using `src:url`) or by its system font name (by using `src:local`). This property is required. For more information, see "Locating embedded fonts" on page 770.

The `fontFamily` property sets the alias for the font that you use to apply the font in style sheets. This property is required.

The `fontStyle` and `fontWeight` properties set the type face values for the font. These properties are optional. The default values are `normal`.

The `flashType` property determines whether to include the FlashType hinting information when embedding the font. This property is optional. The default value is `true`. You cannot use this option when embedding fonts from a SWF file (see "Embedding fonts from SWF files" on page 791). For more information on using FlashType hinting, see "Using FlashType hinting" on page 775.

The following example embeds the MyriadWebPro.ttf font file:

```
@font-face {
  src:url("../assets/MyriadWebPro.ttf");
```

```
  fontFamily: myFontFamily;
  flashType: true;
}
```

The following example embeds that same font, but by using its system font name:

```
@font-face {
  src:local("Myriad Web Pro");
  fontFamily: myFontFamily;
  flashType: true;
}
```

After you embed a font with an `@font-face` declaration, you can use the new `fontFamily` name, or *alias*, in a type or class selector. The following example uses the `myFontFamily` embedded `fontFamily` as a type selector for VBox controls:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFace.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontFamily;
        flashType: true;
     }

     VBox {
        fontFamily: myFontFamily;
     }
  </mx:Style>

  <mx:Panel title="Embedded Font Applied With Type Selector">
     <mx:VBox>
        <mx:Button label="Click Me"/>
        <mx:Label text="Label"/>
        <mx:TextArea width="400" height="75" text="The text
            uses the myClass class selector."
        />
     </mx:VBox>
  </mx:Panel>
</mx:Application>
```

You can also apply the embedded font inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFaceInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontFamily;
        flashType: true;
     }
  </mx:Style>

  <mx:Panel title="Embedded Font Applied Inline">
     <mx:VBox fontFamily="myFontFamily">
        <mx:Button label="Click Me"/>
        <mx:Label text="Label"/>
        <mx:TextArea width="400" height="75" text="This text
            is in Myriad Web Pro."
        />
     </mx:VBox>
  </mx:Panel>
</mx:Application>
```

When you run this example, you might notice that the Button control's label does not appear. This is because the label of a Button control uses a bold typeface. However, the embedded font's typeface (MyriadWebPro) does not contain a definition for the bold typeface. You must *also* embed a font's bold typeface so that the label of a Button control is rendered with the correct font. For information on embedding bold typefaces, see "Using multiple typefaces" on page 781.

Locating embedded fonts

The `src` attribute in the `@font-face` declaration specifies the location of the `fontFamily`. You can specify a `url` or a `local` function. The following table describes these functions:

Attribute	Description
`url`	Embeds a TrueType font by location by specifying a valid URI to the font. The URI can be relative (for example, ../fontfolder/akbar.ttf) or absolute (for example, c:/myfonts/akbar.ttf).

Attribute	Description
`local`	Embeds a locally accessible TrueType font by name rather than location. You use the font name, and not the filename, for the font. For example, you specify "Akbar Bold Italic" rather than "AkbarBI.ttf". You can embed fonts that are locally accessible by the application server's Java Runtime Environment (JRE). These fonts include the *.ttf files in the jre/lib/fonts folder, fonts that are mapped in the jre/lib/font.properties file, and fonts that are made available to the JRE by the operating system (OS). On Windows, TTF files in the /windows/fonts directory (or /winnt/fonts) are available to the local function. On Solaris or Linux, fonts that are registered with a font server, such as xfs, are available. The font name that you specify is determined by the operating system. In general, you do not include the font file's extension, but this is OS-dependent. For more information, consult your operating system documentation.

You must specify the `url` or `local` function of the `src` descriptor in the `@font-face` declaration. All other descriptors are optional. In general, you should use `url` rather than `local`, because pointing to a file is more reliable than using a reference controlled by the operating system.

Do not mix embedded and nonembedded fonts in the same `fontFamily` descriptor.

Embedding fonts in ActionScript

You can embed TTF font files or system fonts by location or by name by using the `[Embed]` metadata tag in ActionScript. To embed a font by location, you use the `source` property in the `[Embed]` metadata tag. To embed a font by name, you use the `systemFont` property in the `[Embed]` metadata tag.

The following examples embed fonts by location by using the [Embed] tag syntax:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFaceActionScriptByLocation.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .mystyle1 {
        fontFamily:myMyriadFont;
        fontSize: 32pt;
     }
     .mystyle2 {
        fontFamily:myBoldMyriadFont;
        fontSize: 32pt;
        fontWeight: bold;
     }
  </mx:Style>

  <mx:Script>
     /*
      * Embed a font by location.
      */
     [Embed(source='../assets/MyriadWebPro.ttf',
        fontName='myMyriadFont',
        mimeType='application/x-font'
     )]
     // You do not use this variable directly. It exists so that
     // the compiler will link in the font.
     private var font1:Class;

     /*
      * Embed a font with bold typeface by location.
      */
     [Embed(source='../assets/MyriadWebPro-Bold.ttf',
        fontWeight='bold',
        fontName='myBoldMyriadFont',
        mimeType='application/x-font',
        flashType='true'
     )]
     private var font2:Class;

  </mx:Script>

  <mx:Panel title="Embedded Fonts Using ActionScript">
     <mx:VBox>
        <mx:Label
            width="100%"
            height="75"
            styleName="mystyle1"
            text="The text uses the MyriadWebPro font."
            rotation="15"
        />
```

```
        <mx:Label
            width="100%"
            height="75"
            styleName="mystyle2"
            text="The text uses the MyriadWebPro-Bold font."
            rotation="15"
        />
    </mx:VBox>
  </mx:Panel>
</mx:Application>
```

You use the value of the `fontName` property that you set in the `[Embed]` tag as the alias (`fontFamily`) in your style definition.

To embed a font with a different typeface (such as bold or italic), you specify the `fontWeight` or `fontStyle` attributes in the `[Embed]` statement and in the style definition. For more information on embedding different typefaces, see "Using multiple typefaces" on page 781.

To show you that the example runs correctly, the Label controls are rotated. If the fonts were not correctly embedded, the text in the Label controls would disappear when you rotated the text.

To embed local or system fonts, you use the system font name rather than the filename for the font. You also specify that name using the `systemFont` attribute in the `[Embed]` tag rather than the source attribute. Otherwise, you use the same syntax, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFaceActionScriptByName.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .mystyle1 {
        fontFamily:myPlainFont;
        fontSize: 32pt;
     }
     .mystyle2 {
        fontFamily:myItalicFont;
        fontSize: 32pt;
        fontStyle: italic;
     }
  </mx:Style>

  <mx:Script>
     /*
      * Embed a font by name.
      */
     [Embed(systemFont='Myriad Web Pro',
        fontName='myPlainFont',
        mimeType='application/x-font'
     )]
     // You do not use this variable directly. It exists so that
     // the compiler will link in the font.
     private var font1:Class;

     /*
      * Embed a font with italic typeface by name.
      */
     [Embed(systemFont='Myriad Web Pro',
        fontStyle='italic',
        fontName='myItalicFont',
        mimeType='application/x-font',
        flashType='true'
     )]
     private var font2:Class;

  </mx:Script>

  <mx:Panel title="Embedded Fonts Using ActionScript">
     <mx:VBox>
        <mx:Label
            width="100%"
            height="75"
```

```
            styleName="mystyle1"
            text="The text uses the plain typeface."
            rotation="15"
        />
        <mx:Label
            width="100%"
            height="75"
            styleName="mystyle2"
            text="The text uses the italic typeface."
            rotation="15"
        />
    </mx:VBox>
  </mx:Panel>
</mx:Application>
```

TIP To get the system font name in Windows, install the Font properties extension. Then right-click on the font's file in Windows Explorer and select the Names tab. Use the value under Font Family Name as the `systemFont` value.

You can specify a subset of the font's character range by specifying the `unicodeRange` parameter in the `[Embed]` metadata tag or the `@font-face` declaration. Embedding a range of characters rather than using the default of all characters can reduce the size of the embedded font and, therefore, reduce the final output size of your SWF file. For more information, see "Setting character ranges" on page 786.

Using FlashType hinting

When you embed fonts, you can use FlashType hinting to provide those fonts with additional information about the font. Embedded fonts that use the FlashType hinting information are typically clearer and appear sharper at smaller font sizes.

By default, fonts that you embed in Flex applications use the FlashType hinting information. This default is set by the `fonts.flash-type` compiler option in the flex-config.xml file (the default value is `true`). You can override this default value by setting the value in your style sheets or changing it in the configuration file. To disable FlashType hinting in style sheets, you set the `flashType` style property to `false` in your `@font-face` rule, as the following example shows:

```
@font-face {
  src:url("../assets/MyriadWebPro.ttf");
  fontFamily: myFontFamily;
  flashType: false;
}
```

Using FlashType hinting can degrade the performance of your compiler. This is not a run-time concern, but can be noticeable if you compile your applications frequently or use the request-time compiler supported by Flex Data Services. Using FlashType hinting can also cause a slight delay when loading SWF files. You notice this delay especially if you are using several different character sets, so be aware of the number of fonts that you use. The presence of FlashType hinting information may also cause an increase in the memory usage in Flash Player. Using four or five fonts, for example, can increase memory usage by approximately 4 MB.

When you embed fonts that use FlashType hinting in your Flex applications, the fonts function exactly as other embedded fonts. They are anti-aliased, you can rotate them, and you can make them partially or wholly transparent.

Font definitions that use FlashType hinting supports several additional styles properties: `fontAntiAliasType`, `fontGridFitType`, `fontSharpness`, and `fontThickness`. These properties are all inheriting styles.

Because the FlashType-related style properties are CSS styles, you can use them in the same way that you use standard style properties, such as `fontFamily` and `fontSize`. For example, a text-based component could use subpixel-fitted FlashType hinting of New Century 14 at sharpness 50 and thickness -35, while all Buttons could use pixel-fitted FlashType hinting of Tahoma 10 at sharpness 0 and thickness 0. These styles apply to all the text in a TextField; you cannot apply them to some characters and not others.

The default values for the FlashType styles properties are defined in the defaults.css file. If you replace this file or use another style sheet that overrides these properties, Flash Player uses the standard font renderer to render the fonts that use FlashType hinting. If you embed fonts that use FlashType hinting, you must set the `fontAntiAliasType` property to `advanced`, or you lose the benefits of the FlashType hinting information.

The following table describes these properties:

Style property	Description
`fontAntiAliasType`	Sets the `antiAliasType` property of internal TextFields. The valid values are `normal` and `advanced`. The default value is `advanced`, which enables the FlashType hinting for the font. Set this property to `normal` to prevent the compiler from using FlashType hinting. This style has no effect for system fonts or fonts embedded without the FlashType hinting information.
`fontGridFitType`	Sets the `gridFitType` property of internal TextFields. The valid values are `none`, `pixel`, and `subpixel`. The default value is `pixel`. For more information, see the TextField and GridFitType classes in *Adobe Flex 2 Language Reference*. This property has the same effect as the `gridFitType` style property of the TextField control for system fonts, only it applies when you embed fonts with FlashType hinting. Changing the value of this property has no effect unless the `fontAntiAliasType` property is set to `advanced`.
`fontSharpness`	Sets the `sharpness` property of internal TextFields. The valid values are numbers from -400 to 400. The default value is 0. This property has the same effect as the `fontSharpness` style property on the TextField control for system fonts, only it applies when you embed fonts with FlashType hinting. Changing the value of this property has no effect unless the `fontAntiAliasType` property is set to `advanced`.
`fontThickness`	Sets the `thickness` property of internal TextFields. The valid values are numbers from -200 to 200. The default value is 0. This property has the same effect as the `fontThickness` style property on the TextField control for system fonts, only it applies when you embed fonts with FlashType hinting. Changing the value of this property has no effect unless the `fontAntiAliasType` property is set to `advanced`.

Detecting embedded fonts

You can use the SystemManager's `isFontFaceEmbedded()` method to determine if the font is embedded or if it has been registered globally with the `register()` method of the Font class. The `isFontFaceEmbedded()` method takes a single argument: the object that describes the font's TextFormat; and returns a Boolean value that indicates whether the font family you specify is embedded, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/DetectingEmbeddedFonts.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
     creationComplete="determineIfFontFaceIsEmbedded()">
  <mx:Style>
     @font-face {
        src: url(../assets/MyriadWebPro.ttf);
        fontFamily: myPlainFont;
        flashType: true;
     }

     .myStyle1 {
        fontFamily: myPlainFont;
        fontSize:12pt
     }
  </mx:Style>
  <mx:Script><![CDATA[
     import mx.managers.SystemManager;
     import flash.text.TextFormat;

     public function determineIfFontFaceIsEmbedded():void {
        var tf1:TextFormat = new TextFormat();
        tf1.font = "myPlainFont";

        var tf2:TextFormat = new TextFormat();
        tf2.font = "Arial";

        var b1:Boolean = Application.application.systemManager.
           isFontFaceEmbedded(tf1);
        var b2:Boolean = Application.application.systemManager.
           isFontFaceEmbedded(tf2);
        l1.text = tf1.font + " (" + b1 + ")";
        l2.text = tf2.font + " (" + b2 + ")";
     }
  ]]></mx:Script>
  <mx:Text id="text1" styleName="myStyle1" text="Rotate Me"/>

  <mx:Button label="Rotate +1" click="++text1.rotation;"/>
  <mx:Button label="Rotate -1" click="--text1.rotation;"/>

  <mx:Form>
```

```
    <mx:FormItem label="isFontFaceEmbedded:">
       <mx:Label id="l1"/>
    </mx:FormItem>
    <mx:FormItem label="isFontFaceEmbedded:">
       <mx:Label id="l2"/>
    </mx:FormItem>
  </mx:Form>
</mx:Application>
```

You can use the Font class's `enumerateFonts()` method to output information about device or embedded fonts. The following example lists embedded fonts:

```
<?xml version="1.0"?>
<!-- fonts/EnumerateFonts.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="listFonts()">
  <mx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFont;
        flashType: true;
     }

     @font-face {
        src:url("../assets/MyriadWebPro-Bold.ttf");
        fontFamily: myFont;
        fontWeight: bold;
        flashType: true;
     }

     @font-face {
        src:url("../assets/MyriadWebPro-Italic.ttf");
        fontFamily: myFont;
        fontStyle: italic;
        flashType: true;
     }

     .myPlainStyle {
        fontSize: 32;
        fontFamily: myFont;
     }

     .myBoldStyle {
        fontSize: 32;
        fontFamily: myFont;
        fontWeight: bold;
     }

     .myItalicStyle {
        fontSize: 32;
        fontFamily: myFont;
        fontStyle: italic;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     private function listFonts():void {
        var fontArray:Array = Font.enumerateFonts(false);
        for(var i:int = 0; i < fontArray.length; i++) {
```

```
            var thisFont:Font = fontArray[i];
            if (thisFont.fontType == "embedded") {
             trace("name: " + thisFont.fontName +
             "; typeface: " + thisFont.fontStyle +
             "; type: " + thisFont.fontType);
            }
         }
      }
   ]]></mx:Script>

   <mx:VBox>
      <mx:Label text="Plain Label" styleName="myPlainStyle"/>
      <mx:Label text="Italic Label" styleName="myBoldStyle"/>
      <mx:Label text="Bold Label" styleName="myItalicStyle"/>
   </mx:VBox>

</mx:Application>
```

The following list shows the output:

```
name: myFont; typeface: regular; type: embedded
name: myFont; typeface: bold; type: embedded
name: myFont; typeface: italic; type: embedded
```

The `enumerateFonts()` method takes a single Boolean argument: `enumerateDeviceFonts`. The default value of the `enumerateDeviceFonts` property is `false`, which means it returns an Array of embedded fonts by default.

If you set the `enumerateDeviceFonts` argument to `true`, the `enumerateFonts()` method returns an Array of available device fonts on the client system, but only if the client's mms.cfg file sets the `DisableDeviceFontEnumeration` property to 0, the default value. If you set `DisableDeviceFontEnumeration` property to 1, Flash Player cannot list device fonts on a client computer unless you explicitly configure the client to allow it. For more information about configuring the client with the mms.cfg file, see the Flash Player documentation.

Using multiple typefaces

Most fonts have four typeface styles: plain, boldface, italic, and bold-italic. You can embed any number of typeface styles in your Flex applications. If you only embed the boldface typeface in your application, you cannot use the normal (or plain) typeface unless you also embed that typeface. For each typeface that you use, you must add a new `@font-face` declaration to your style sheet.

Some Flex controls, such as Button, use the boldface typeface style by default, rather than the plain style. If you use an embedded font for a Button label, you must embed the boldface font style for that font.

The following example embeds the boldface, oblique, and plain typefaces of the Myriad Web Pro font. After you define the font face, you define selectors for the font by using the same alias as the `fontFamily`. You define one for the boldface, one for the oblique, and one for the plain face. To apply the font styles, this example applies the class selectors to the Label controls inline:

```
<?xml version="1.0"?>
<!-- fonts/MultipleFaces.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFont;
        flashType: true;
     }

     @font-face {
        /* Note the different filename for boldface. */
        src:url("../assets/MyriadWebPro-Bold.ttf");
        fontFamily: myFont; /* Notice that this is the same alias. */
        fontWeight: bold;
        flashType: true;
     }

     @font-face {
        /* Note the different filename for italic face. */
        src:url("../assets/MyriadWebPro-Italic.ttf");
        fontFamily: myFont; /* Notice that this is the same alias. */
        fontStyle: italic;
        flashType: true;
     }

     .myPlainStyle {
        fontSize: 32;
        fontFamily: myFont;
     }

     .myBoldStyle {
        fontSize: 32;
        fontFamily: myFont;
        fontWeight: bold;
     }

     .myItalicStyle {
        fontSize: 32;
        fontFamily: myFont;
        fontStyle: italic;
     }
  </mx:Style>
```

```
  <mx:VBox>
     <mx:Label text="Plain Label" styleName="myPlainStyle"/>
     <mx:Label text="Italic Label" styleName="myItalicStyle"/>
     <mx:Label text="Bold Label" styleName="myBoldStyle"/>
  </mx:VBox>

</mx:Application>
```

Optionally, you can apply the boldface or oblique type to controls inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/MultipleFacesAppliedInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFont;
        flashType: true;
     }

     @font-face {
        src:url("../assets/MyriadWebPro-Bold.ttf");
        fontFamily: myFont;
        fontWeight: bold;
        flashType: true;
     }

     @font-face {
        src:url("../assets/MyriadWebPro-Italic.ttf");
        fontFamily: myFont;
        fontStyle: italic;
        flashType: true;
     }

     .myStyle1 {
        fontSize: 32;
        fontFamily: myFont;
     }
  </mx:Style>

  <mx:VBox styleName="myStyle1">
     <mx:Label text="Plain Label"/>
     <mx:Label text="Italic Label" fontStyle="italic"/>
     <mx:Label text="Bold Label" fontWeight="bold"/>
  </mx:VBox>

</mx:Application>
```

If you embed a system font by name rather than by a font file by location, you use the font's family name (such as Myriad Web Pro) in the `@font-face` rule and not the typeface name (such as MyriadWebPro-Bold). The following example embeds a plain typeface and an italic typeface by using the system font names:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontFaceCSSByName.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     @font-face {
        src:local("Myriad Web Pro");
        fontFamily: myPlainFont;
        flashType: true;
     }

     @font-face {
        src:local("Myriad Web Pro");
        fontFamily: myItalicFont;
        fontStyle: italic;
        flashType: true;
     }

     .mystyle1 {
        fontFamily:myPlainFont;
        fontSize: 32pt;
     }
     .mystyle2 {
        fontFamily:myItalicFont;
        fontSize: 32pt;
        fontStyle: italic;
     }
  </mx:Style>

  <mx:Panel title="Embedded Fonts Using CSS">
     <mx:VBox>
        <mx:Label
            width="100%"
            height="75"
            styleName="mystyle1"
            text="The text uses the plain typeface."
            rotation="15"
        />
        <mx:Label
            width="100%"
            height="75"
            styleName="mystyle2"
            text="The text uses the italic typeface."
            rotation="15"
        />
     </mx:VBox>
```

```
  </mx:Panel>
</mx:Application>
```

Flex also supports using the `<b>` and `<i>` HTML tags in text-based controls, such as Label, to apply the boldface or italic font to the text.

If you use a bold-italic font, the font must have a separate typeface for that font. You specify both descriptors (`fontWeight` and `fontStyle`) in the `@font-face` and selector blocks, as the following example shows:

```
@font-face {
  src:url("../assets/KNIZIA-BI.TTF");
  fontStyle: italic;
  fontWeight: bold;
  fontFamily: myFont;
  flashType: true;
}
.myBoldItalicStyle {
  fontFamily:myFont;
  fontWeight:bold;
  fontStyle:italic;
  fontSize: 32;
}
```

In the `@font-face` definition, you can specify whether the font is boldface or italic font by using the `fontWeight` and `fontStyle` descriptors. For a bold font, you can set `fontWeight` to `bold` or an integer greater than or equal to 700. You can specify the `fontWeight` as `plain` or `normal` for a nonboldface font. For an italic font, you can set `fontStyle` to `italic` or `oblique`. You can specify the `fontStyle` as `plain` or `normal` for a nonitalic face. If you do not specify a `fontWeight` or `fontStyle`, Flex assumes you embedded the plain or regular font face.

You can also add any other descriptors for the embedded font, such as `fontSize`, to the selector, as you would with any class or type selector.

By default, Flex includes the entire font definition for each embedded font in the application, so you should limit the number of fonts that you use to reduce the size of the application. You can limit the size of the font definition by defining the character range of the font. For more information, see "Setting character ranges" on page 786.

About the font managers

Flex includes two font managers by default. They take the embedded TrueType font definitions and draw each character in Flash Player. The two font managers are Batik and JRE, represented by the BatikFontManager and JREFontManager classes, respectively. The Batik font manager is limited because it cannot use local fonts; but, in general, it provides smoother rendering and more accurate line metrics (which effects multiline text and line-length calculations). Local fonts are fonts that are accessible through the operating system. The JRE font manager, on the other hand, can access local fonts used by the operating system, but the quality of output is generally not as good.

You determine which font manager you use in the flex-config.xml file. The default setting is to use both, as the following example shows:

```
<fonts>
  <managers>
    <manager-class>flash.fonts.JREFontManager</manager-class>
    <manager-class>flash.fonts.BatikFontManager</manager-class>
  </managers>
</fonts>
```

The preference of `<manager>` elements is in reverse order. This means that by default the Batik font manager is the preferred font manager. However, the Batik font manager does not support system fonts, therefore, the JRE font manager is also available to handle these cases.

Setting character ranges

By specifying a range of symbols that compose the face of an embedded font, you reduce the size of an embedded font. Each character in a font must be described; if you remove some of these characters, it reduces the overall size of the description information that Flex must include for each embedded font.

You can set the range of glyphs in the flex-config.xml file or in the `font-face` declaration in each MXML file. You specify individual characters or ranges of characters using the Unicode values for the characters, and you can set multiple ranges for each font declaration.

If you use a character that is outside of the declared range, Flex displays nothing for that character.

The techniques in this section focus on setting character ranges in Flex applications. However, if you embed a font from a SWF file, you can restrict the character range in Flash. For more information, see "Embedding fonts from SWF files" on page 791.

For more information on character ranges, see the CSS-2 Fonts specification at www.w3.org/TR/1998/REC-CSS2-19980512/fonts.html#descdef-unicode-range.

Setting ranges in font-face declarations

You can set the range of allowable characters in an MXML file by using the `unicodeRange` attribute of the `font-face` declaration. The following example embeds the Myriad Web Pro font and defines the range of characters for the font in the `<mx:Style>` tag:

```
<?xml version="1.0"?>
<!-- fonts/EmbeddedFontCharacterRange.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     @font-face {
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily: myFontFamily;
        flashType: true;
        unicodeRange:
           U+0041-U+005A, /* Upper-Case [A..Z] */
           U+0061-U+007A, /* Lower-Case a-z */
           U+0030-U+0039, /* Numbers [0..9] */
           U+002E-U+002E; /* Period [.] */
     }

     TextArea {
        fontFamily: myFontFamily;
        fontSize: 32;
     }
  </mx:Style>

  <mx:Panel title="Embedded Font Character Range">
        <mx:TextArea
            width="400"
            height="150"
            text="The Text Uses Only Some of Available
            Characters 0 1 2 3 4 5 6 7 8 9."
        />
  </mx:Panel>
</mx:Application>
```

Setting ranges in flex-config.xml

You can specify the language and character range for embedded fonts in the flex-config.xml file by using the `<language-range>` child tag. This lets you define the range once and use it across multiple `@font-face` blocks.

The following example creates an `englishRange` and an `otherRange` named ranges in the flex-config.xml file:

```
<fonts>
  <languages>
    <language-range>
```

```
      <lang>englishRange</lang>
      <range>U+0020-U+007E</range>
    </language-range>
    <language-range>
      <lang>otherRange</lang>
      <range>U+00??</range>
    </language-range>
  <languages>
</fonts>
```

In your MXML file, you point to the defined ranges by using the `unicodeRange` attribute of the `@font-face` declaration, as the following example shows:

```
@font-face {
  fontFamily: myPlainFont;
  src: url("../assets/MyriadWebPro.ttf");
  flashType: true;
  unicodeRange: "englishRange";
}
```

Flex includes a file that lists convenient mappings of the Flash UnicodeTable.xml character ranges for use in the Flex configuration file. For Adobe Flex Data Services, the file is located at *flex_app_root*/WEB-INF/flex/flash-unicode-table.xml; for Adobe Flex SDK, the file is located at *flex_install_dir*/frameworks/flash-unicode-table.xml.

The following example shows the predefined range Latin 1:

```
<language-range>
  <lang>Latin I</lang>
  <range>U+0020,U+00A1-U+00FF,U+2000-U+206F,U+20A0-U+20CF,U+2100-U+2183
  </range>
</language-range>
```

To make ranges listed in the flash-unicode-table.xml file available in your Flex applications, copy the ranges from this file and add them to the flex-config.xml files.

Detecting available ranges

You can use the Font class to detect the available characters in a font. You do this with the `hasGlyphs()` method.

The following example embeds the same font twice, restricting each to a different character ranges. The first font only includes support for the letters A and B. The second font family includes all 128 glyphs in the Basic Latin block.

```
<?xml version="1.0"?>
<!-- charts/CharacterRangeDetection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="checkCharacterSupport();">
  <mx:Style>
     @font-face{
        font-family: myABFont;
        flashType: true;
        src:url("../assets/MyriadWebPro.ttf");
        /* Limit range to the letters A and B. */
        unicodeRange: U+0041-U+0042;
     }
     @font-face{
        font-family: myWideRangeFont;
        flashType: true;
        src:url("../assets/MyriadWebPro.ttf");
        /* Set range to the 128 characters in
        the Basic Latin block. */
        unicodeRange: U+0041-U+007F;
     }
  </mx:Style>
  <mx:Script><![CDATA[
     public function checkCharacterSupport():void {
        var fontArray:Array = Font.enumerateFonts(false);
        for(var i:int = 0; i < fontArray.length; i++) {
           var thisFont:Font = fontArray[i];
           if (thisFont.hasGlyphs("DHARMA")) {
            trace(thisFont.fontName +
                " supports these glyphs");
           } else {
            trace(thisFont.fontName +
            " does not support these glyphs");
           }
        }
     }
  ]]></mx:Script>
</mx:Application>
```

Embedding double-byte fonts

When using double-byte fonts in Flex, you should embed the smallest possible set of characters. If you embed a font's entire character set, the size of your application's SWF file can be very large. You can define sets of Unicode character ranges in the flex-config.xml file, and then reference the name of that range in your style's `@font-face` declaration.

Flex provides predefined character ranges for common double-byte languages such as Thai, Kanji, Hangul, and Hebrew in the flash-unicode-table.xml file. This file is not processed by Flex, but is included to provide you with ready definitions for various character ranges. For example, the following character range for Thai is listed in the flash-unicode-table.xml file:

```
<language-range>
  <lang>Thai</lang>
  <range>U+0E01-U+0E5B</range>
</language-range>
```

To use this language in your Flex application, copy the character range to the flex-config.xml file or pass it on the command line by using the `fonts.languages.language-range` option. Add the full definition as a child tag to the `<languages>` tag, as the following example shows:

```
<flex-config>
  <fonts>
    <languages>
      <language-range>
        <lang>thai</lang>
        <range>U+0E01-U+0E5B</range>
      </language-range>
    </languages>
  </fonts>
  ...
</flex-config>
```

You can change the value of the `<lang>` element to anything you want. When you embed the font by using CSS, you refer to the language by using this value in the `unicodeRange` property of the `@font-face` declaration, as the following example shows:

```
@font-face {
  fontFamily:"Thai_font";
  src: url("../assets/THAN.TTF"); /* Embed from file */
  flashType: true;
  unicodeRange:"thai"
}
```

Embedding fonts from SWF files

You can embed fonts that are embedded in SWF files into your Flex applications.

About embedding fonts from SWF files

When you embed a font in a SWF file that you use in your Flex application, you can embed any font that is supported by Flash 8. This includes Type 1 PostScript and bitmap (Macintosh only) fonts, as well as fonts with FlashType hinting. To do this, you create a SWF file in Flash 8 that contains the fonts you want. You then reference that SWF file's font contents so that it is embedded into your Flex application.

In general, you should include the four primary typefaces for each font that you want to embed (plain, bold, italic, and bold italic). You should do this even if the font you are embedding does not have separate TTF files for each typeface. One reason to do this is that some Flex controls use one of the non-plain typefaces.

For example, the label on a Button control uses the bold typeface of the font that it uses. If you embedded a typeface and tried to use it on a Button, it would not be applied because the Button control requires the bold typeface.

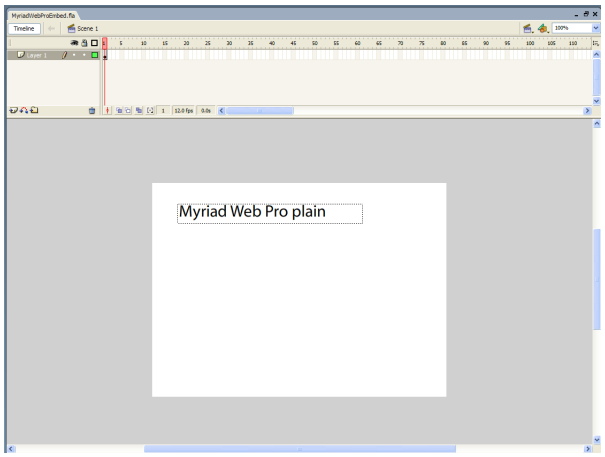
Creating Flash 8 files with embedded fonts

The first step in embedding fonts from SWF files in your Flex applications is to create a SWF file in Flash that contains those fonts. After you complete the steps in this section, use the instructions in "Embedding fonts from SWF files into Flex applications" on page 793 to complete the process.

To create a Flash 8 SWF file that embeds fonts:

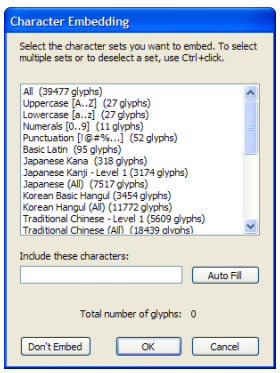
1. Create a document in Flash 8.
2. Select the Text Tool, and create a text area on the Stage by dragging the mouse.
3. In the Properties panel, select the font you want to embed from the font drop-down list.

4. Enter at least one character of text in the text area. If you are embedding more than one font, it is helpful to enter the name of the font and its typeface; for example:



5. In the Properties panel, select Dynamic Text drop-down list to make this text dynamic.
6. If you use FlashType hinting in your fonts, ensure that either the Anti-Alias for Readability or Custom Anti-Alias option is the selected anti-aliasing mode. If you select any other option, Flash does not include FlashType hinting information with your fonts' SWF file. For more information about using FlashType hinting, see "Using FlashType hinting" on page 775.
7. In the Properties panel, click the Embed button.

   The Character Embedding dialog box appears:



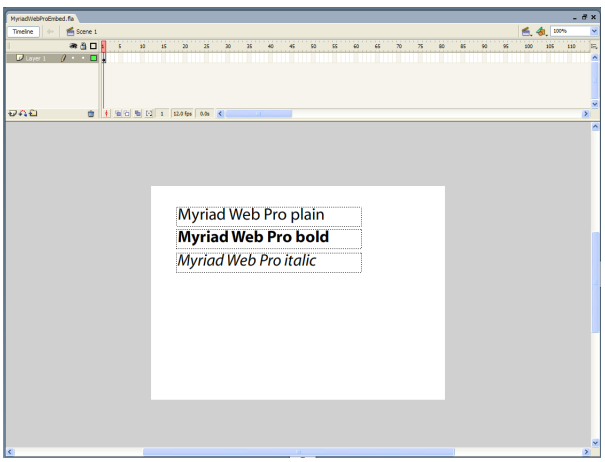
8. Select one or more ranges to use. Select only the ranges that are necessary.

You should select All only if it is absolutely necessary. The more glyphs that you add to the SWF file, the greater its size. For example, if you select All for Century Gothic, the final SWF file size is about 270 KB. If you select Uppercase, Lowercase, and Punctuation only, the final SWF file size is about 14 KB.

Do not select Auto Fill unless you know exactly which characters you are going to use in your Flex application.

9. Create additional text-based controls, one for each of the typefaces you want (such as bold, and italic). For the bold control, apply the bold typeface. For the italic control, apply the italic typeface. For the bold and italic control (if applicable), apply the bold and italic typefaces. For each typeface, select the range of glyphs to include in the SWF file.

   Your final Stage might appear like the following example:



10. Select File > Export > Export Movie.

    Flash generates a SWF file that you import into your Flex application.

Embedding fonts from SWF files into Flex applications

This section describes the second of two steps in the process for embedding fonts from SWF files from a Flash file into your Flex applications. Before performing the steps in this section, you must create a SWF file in Flash that embeds the fonts. For more information, see "Creating Flash 8 files with embedded fonts" on page 791.

To embed fonts from a Flash 8 SWF file into your Flex application:

1. Embed each typeface with an `@font-face` declaration, as the following example shows:

```
/* assets/FlashTypeStyles.css */
@font-face {
   src:url("../assets/MyriadWebProEmbed.swf");
   fontFamily: "Myriad Web Pro";
}
@font-face {
   src:url("../assets/MyriadWebProEmbed.swf");
   fontFamily: "Myriad Web Pro";
   fontWeight: bold;
}
@font-face {
   src:url("../assets/MyriadWebProEmbed.swf");
   fontFamily: "Myriad Web Pro";
   fontStyle: italic;
}
```

   You specify the location of the SWF file by using the `src` attribute. You set the value of the `fontFamily` attribute to the name of the font as it appears in the list of available fonts in Flash. You must specify a new `@font-face` entry for each of the typeface properties if the font face is not plain. For more information on using the `@font-face` declaration, see "Embedded font syntax" on page 768.

   Do not specify a value for the `flashType` property in the `@font-face` declaration. This is because anti-aliasing settings in Flash determine whether to include the FlashType hinting information. After a SWF file that contains fonts has been generated, you cannot add the hinting information to the SWF file by using the Flex compiler or from within the Flex application. For more information on using FlashType hinting, see "Using FlashType hinting" on page 775.

2. Define a style for each embedded font typeface. You can define this style as an external style sheet or in a `<mx:Style>` block, as the following example shows:

```
/* assets/FlashTypeClassSelectors.css */
.myPlainStyle {
    fontFamily: "Myriad Web Pro";
    fontSize: 24;
}
.myItalicStyle {
    fontFamily: "Myriad Web Pro";
    fontSize: 24;
    fontStyle: italic;
}
.myBoldStyle {
    fontFamily: "Myriad Web Pro";
    fontSize: 24;
    fontWeight: bold;
}
```

You must specify the `fontFamily` property in the style definition, and it must match the `fontFamily` property set in the `@font-face` declaration. Regardless of which typeface you are defining, the value of the `fontFamily` property is the same. For example, if the typeface is boldface, you do not set the `fontFamily` property to "Myriad Web Pro Bold", but just "Myriad Web Pro."

You must also specify all typeface properties in the style definition, just as you did in the `@font-face` declaration.

3. Apply the new style to your Flex controls, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/EmbedFlashTypeFonts.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style source="../assets/FlashTypeStyles.css"/>
  <mx:Style source="../assets/FlashTypeClassSelectors.css"/>

  <mx:Panel title="Embedded Font">
     <mx:VBox>
        <mx:Label text="Plain Label" styleName="myPlainStyle"/>
        <mx:Label text="Italic Label" styleName="myBoldStyle"/>
        <mx:Label text="Bold Label" styleName="myItalicStyle"/>
     </mx:VBox>
  </mx:Panel>
</mx:Application>
```

You can also define and apply the styles inline, rather than define them in a style sheet and apply them with the `styleName` property. The following example sets the value of the `fontFamily` and `fontStyle` properties inline to apply the Myriad Web Pro font's italic and bold typefaces to the Label controls:

```
<?xml version="1.0"?>
<!-- fonts/EmbedFlashTypeFontsInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style source="../assets/FlashTypeStyles.css"/>

  <mx:Panel title="Embedded FlashType Font">
     <mx:VBox>
        <mx:Label text="Plain Label" fontFamily="Myriad Web Pro"
fontSize="24"/>
        <mx:Label text="Italic Label" fontFamily="Myriad Web Pro"
fontStyle="italic" fontSize="24"/>
        <mx:Label text="Bold Label" fontFamily="Myriad Web Pro"
fontWeight="bold" fontSize="24"/>
     </mx:VBox>
  </mx:Panel>
</mx:Application>
```

The following example embeds the Myriad Web Pro font into the Flex application. It embeds each typeface, defines the styles for them, and applies those styles to Text controls.

```
<?xml version="1.0"?>
<!-- fonts/EmbedFlashTypeFontsFull.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     @font-face {
        src:url("../assets/MyriadWebProEmbed.swf");
        fontFamily: "Myriad Web Pro";
     }
     @font-face {
        src:url("../assets/MyriadWebProEmbed.swf");
        fontFamily: "Myriad Web Pro";
        fontWeight: bold;
     }
     @font-face {
        src:url("../assets/MyriadWebProEmbed.swf");
        fontFamily: "Myriad Web Pro";
        fontStyle: italic;
     }
     .myPlainStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
     }
     .myItalicStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
        fontStyle: italic;
     }
     .myBoldStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
        fontWeight: bold;
     }
  </mx:Style>

  <mx:Panel title="Embedded SWF-based Font">
     <mx:VBox>
        <!-- Apply each custom style to Text controls. -->
        <mx:Text id="text1" styleName="myPlainStyle" text="Plain Text"/>
        <mx:Text id="text2" styleName="myBoldStyle" text="Bold Text"/>
        <mx:Text id="text3" styleName="myItalicStyle" text="Italic Text"/>
     </mx:VBox>
  </mx:Panel>

  <!-- Rotate the Text controls. If the text disappears when the control is
     rotated, then the font is not properly embedded. -->
  <mx:Button label="Rotate +1" click="++text1.rotation;++text2.rotation;
     ++text3.rotation;"/>
```

```
  <mx:Button label="Rotate -1" click="--text1.rotation;--text2.rotation;
     --text3.rotation;"/>
</mx:Application>
```

When you run the example, click the Button controls to rotate the text. This ensures that the fonts were properly embedded; if the fonts were not properly embedded, the text disappears when rotated.

Instead of using the `@font-face` CSS syntax, you can use the `[Embed]` metadata keyword to embed fonts from SWF files in your Flex application. This can give you greater control over the font in ActionScript. To do it, you use the same SWF file that you created in previous examples. In your Flex application, you associate each font face with its own variable, as the following example shows:

```
<?xml version="1.0"?>
<!-- fonts/EmbedFlashTypeFontsActionScriptSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myPlainStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
     }
     .myItalicStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
        fontStyle: italic;
     }
     .myBoldStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
        fontWeight: bold;
     }
  </mx:Style>

  <mx:Script><![CDATA[

        [Embed(source='../assets/MyriadWebProEmbed.swf',
            fontName='Myriad Web Pro'
        )]
        private static var plainFont:Class;

        [Embed(source='../assets/MyriadWebProEmbedWithFontSymbols.swf',
            fontName='Myriad Web Pro',
            fontStyle='italic'
        )]
        private static var italicFont:Class;

        [Embed(source='../assets/MyriadWebProEmbedWithFontSymbols.swf',
            fontName='Myriad Web Pro',
            fontWeight='bold'
```

```
        )]
        private static var boldFont:Class;

    ]]></mx:Script>

    <mx:Panel title="Embedded SWF-Based Font">
       <mx:VBox>
          <!-- Apply each custom style to Text controls. -->
          <mx:Text id="text1" styleName="myPlainStyle" text="Plain Text"/>
          <mx:Text id="text2" styleName="myBoldStyle" text="Bold Text"/>
          <mx:Text id="text3" styleName="myItalicStyle" text="Italic Text"/>
       </mx:VBox>
    </mx:Panel>

    <!-- Rotate the Text controls. If the text disappears when the control is
       rotated, then the font is not properly embedded. -->
    <mx:Button
      label="Rotate +1"
      click="++text1.rotation;++text2.rotation;++text3.rotation;"
    />
    <mx:Button
      label="Rotate -1"
      click="--text1.rotation;--text2.rotation;--text3.rotation;"
    />
</mx:Application>
```

You must define a variable of type Class so that the compiler links the fonts into the final Flex application SWF file. This example sets the value of the static fontPlain, fontBold, and fontItalic variables, but they are not used in the rest of the application.

When you use the `[Embed]` statement to embed fonts in your Flex application, you must still define styles for those fonts so that those styles can be applied to Flex components, as you would if you embedded the fonts with the `@font-face` declarations.

You can also access fonts from SWF files as a Font Symbol.

To use fonts from SWF files that are Font Symbols:

1. In Flash 8, select Window > Library to show the contents of the library. It should be empty.
2. Right-click the mouse when it is over the library, and select New Font to create a new font symbol.
3. In the Font Symbol Properties dialog box, give the new font symbol a name and select the applicable font symbol properties (such as bold or italic). You use the symbol name you specify here in your `[Embed]` statement. Do this for each font typeface (such as plain, bold, italic, and bold italic).
4. After you create a font symbol in the library, right click the symbol in the library and select Linkage.

5. In the Linkage Properties dialog box, select Export for ActionScript, and click OK. The Identifier in this dialog box should match the symbol name you specified in the previous step. Do this for each font symbol in the library.
6. In your Flex application's `[Embed]` statement, point to the font symbol using the symbol attribute. Do not specify any other characteristics of the font such as the MIME type, `fontStyle`, `fontWeight`, or `fontName` in the `[Embed]` statement.

The following example embeds the Myriad Web Pro font that was exported from Flash 8 with font symbols in the library:

```
<?xml version="1.0"?>
<!-- fonts/EmbedFlashTypeFontsActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myPlainStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
     }
     .myItalicStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
        fontStyle: italic;
     }
     .myBoldStyle {
        fontFamily: "Myriad Web Pro";
        fontSize: 24;
        fontWeight: bold;
     }
  </mx:Style>

  <mx:Script><![CDATA[

        [Embed(source='../assets/MyriadWebProEmbedWithFontSymbols.swf',
            symbol='MyriadPlain'
        )]
        private var font1:Class;

        [Embed(source='../assets/MyriadWebProEmbedWithFontSymbols.swf',
            symbol='MyriadBold'
        )]
        private var font2:Class;

        [Embed(source='../assets/MyriadWebProEmbedWithFontSymbols.swf',
            symbol='MyriadItalic'
        )]
        private var font3:Class;

  ]]></mx:Script>

  <mx:Panel title="Embedded SWF-based Font">
     <mx:VBox>
        <!-- Apply each custom style to Text controls. -->
        <mx:Text id="text1"
            styleName="myPlainStyle"
            text="Plain Text"
        />
        <mx:Text id="text2"
```

```
            styleName="myBoldStyle"
            text="Bold Text"
        />
        <mx:Text id="text3"
            styleName="myItalicStyle"
            text="Italic Text"
        />
      </mx:VBox>
  </mx:Panel>

  <!-- Rotate the Text controls. If the text disappears when the control is
     rotated, then the font is not properly embedded. -->
  <mx:Button
    label="Rotate +1"
    click="++text1.rotation;++text2.rotation;++text3.rotation;"
  />
  <mx:Button
    label="Rotate -1"
    click="--text1.rotation;--text2.rotation;--text3.rotation;"
  />
</mx:Application>
```

Troubleshooting

There are some techniques that you can use to successfully embed fonts into your Flex applications.

Resolving compiler errors

The following table describes common compiler errors and their solutions:

Error	Solution
Unable to resolve '*swf_file_name*' for transcoding	Indicates that the font was not found by the compiler. Ensure that the path to the font is correct in the `@font-face` declaration or the `[Embed]` tag, and that the path is accessible by the compiler. If you are using the `local` property to specify the location of the font in the `@font-face` declaration, ensure that the font is locally accessible. Check that it is an entry in the fonts.properties file or that the font is in the system font search path. For more information, see "Embedded font syntax" on page 768.

Error	Solution
Font '*font_name*' with *style_description* not found	Indicates that the `fontName` parameter used in the `[Embed]` statement might not match the name of the font. For fonts in SWF files, ensure that the spelling and word spacing of the font name in the list of available fonts in Flash is the same as the `fontName` attribute in your `[Embed]` statement and the `fontFamily` property that you use in your style definitions. This error can also mean that the font's style was not properly embedded in Flash. Open the FLA file and ensure that there is a text area with the font and style described, that the text is dynamic, and that you selected a character range for that text.

Resolving run-time errors

To ensure that the fonts are properly embedded, try rotating the controls that use the fonts. You can only view rotated fonts if they are embedded, so if the text disappears when you rotate the control, the font is not properly embedded.

To properly embed your fonts, try the following techniques:

- If one type of control is not correctly displaying its text, ensure that you are embedding the appropriate typeface. For example, the Button control's text labels require the bold typeface. If you do not embed the bold typeface, the Button control does not display the embedded font.
- In your Flex application, ensure that you set all properties for each font typeface in the `@font-face` declaration or `[Embed]` statement. To embed a bold typeface, you must set the `fontWeight` property to `bold`, as the following example shows:

```
@font-face {
  src: url(../assets/MyriadWebProEmbed.ttf);
  fontFamily: "Myriad Web Pro";
  fontWeight: bold;
}
```

  You also must set the `fontWeight` style property in your style definition:

```
.myStyle2 {
  fontFamily:"Myriad Web Pro";
  fontWeight:bold;
  fontSize:12pt;
}
```

If you use the `[Embed]` statement, you must set the `fontWeight` property to `bold` as the following example shows:

```
[Embed(source="MyriadWebProEmbed.ttf", fontName="Myriad Web Pro",
  fontWeight="bold")]
```

- For fonts in SWF files, open the FLA file in Flash and ensure that all of the typefaces were added properly. Select each text area and do the following:
  - Check that the font name is correct. Ensure that the spelling and word spacing of the font name in the list of available fonts in Flash is the same as the `fontFamily` property in the `@font-face` declaration or the `fontName` attribute in your `[Embed]` statement. This value must also match the `fontFamily` property that you use in your style definitions.
  - Check that the style is properly applied. For example, select the bold text area and check that the typeface really is bold.
  - Click the Embed button and ensure that the range of embedded characters includes the characters that you use in your Flex application.
  - Check that each text area is set to Dynamic Text and not Static Text or Input Text. The type of text is indicated by the first drop-down box in the text's Properties tab.
- For fonts in SWF files, ensure that you are using the latest SWF file that contains your fonts and was generated in Flash. Regenerate the SWF file in Flash if necessary.

CHAPTER 20

Using Skins

20

This topic describes how to add skins to Adobe Flex components by using ActionScript class files or image files.

Contents

About skinning

Skinning is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of images, SWF files, or class files that contain drawing API methods.

Skins define parts or entire components in various states. For example, a Button control has many skins, each representing a state for the button. These skins include downSkin, upSkin, and overSkin, and represent the way a button appears when it is down, up, and when the mouse is over the button. These three default skins appear as follows:

upSkin overSkin downSkin

Other controls have similar states. For example, RadioButton and RadioButton controls, which derives from Button, also have upSkin, downSkin, and overSkin properties. A control such as ComboBox also has skins the define the appearance of the control when it is in certain states. These skins include disbaledSkin, downSkin, and overSkin. Controls that have no direct hierarchical relationship to Button, such as Accordion, have skins such as borderSkin and focusSkin.

Skins are applied as style properties. You apply them using MXML tag properties, the StyleManager class, `<mx:Style>` blocks, or in style sheets. Most complex skinning uses style sheets to organize and apply the skins to a Flex application. Style sheets can be loaded at compile time or at run time. For information on loading style sheets at run time, see "Loading style sheets at run time" on page 744.

There are two types of skins: graphical and programmatic. *Graphical skins* are images that define the appearance of the skin. These images can JPEG, GIF, or PNG files, or they can be symbols embedded in SWF files.

You draw *programmatic skins* by using ActionScript statements and define these skins in class files. To change the appearance of controls that use programmatic skins, you edit an ActionScript class file and compile it with your Flex application.

One advantage of programmatic skinning is that it provides you with a great deal of control over styles. For example, you cannot control the radius of a Button control's corners with style sheets or graphical skins. You can also develop programmatic skins directly in your Flex authoring environment or any text editor, without using a graphics tool such as Adobe Flash. Programmatic skins also tend to use less memory because they contain no external image files.

About icons

Icons are special style properties that are treated similarly to skins. Icon style properties (such as icon, upIcon, and downIcon) define a visual element added to a control. This visual element does not replace the appearance of the component's skin, but is added to that skin.

For Button controls, icons enhance a button's appearance. They are added to the button and displayed next to the button's label. For CheckBox and RadioButton controls, icons replace the appearance of the checkbox or button on the control. They do not affect the appearance of the control's label.

For list-based controls, icons define the appearance of an item's graphical representation in the list. For containers, icons define a symbol for the container in some navigator containers (such as in the tab of a TabNavigator or the header of an Accordion).

For Buttons, setting the icon property overrides all other icon settings such as upIcon and downIcon.

Resources for skins

Flex 2 includes the following graphical and programmatic source files for skins:

Base skin classes in the mx.skins package These abstract skin classes define the basic functionality of skin classes in Flex. For more information, see "Programmatic skinning" on page 816.

Programmatic Halo skins in the mx.skins.halo package These concrete skin classes extend the base skin classes in the mx.skins package. You can extend or edit these skins to create new programmatic skins based on the default Flex look and feel. For more information, see "Programmatic skinning" on page 816.

HaloClassic skins These skins were used in Flex 1.x. You can retain the look of Flex 1.x applications by using these skins. The HaloClassic.swc file is located in the framework/themes directory. For more information, see "About themes" on page 756.

Graphical Aeon theme The Aeon theme includes the AeonGraphical.css file and the AeonGraphical.swf file that defines the skin symbols. These are in the framework/themes directory. In addition, Flex includes the FLA source file for the AeonGraphical.swf file. For more information, see "About themes" on page 756.

You can use these files to create skins based on the Flex look and feel, or you can create your own by following the instructions in this topic.

For additional information about skinning, see www.adobe.com/go/flex2_skinning.

Graphical skinning

To use graphical skins, you embed image files in your Flex application. Then you define these files as style properties that replace the existing skins of the component.

For example, to change the appearance of a Button control, you might create three image files called orb_up_skin.gif, orb_down_skin.gif, and orb_over_skin.gif:

orb_up_skin.gif

orb_over_skin.gif

orb_down_skin.gif

In your style sheet, you define the values for the `upSkin`, `downSkin`, and `overSkin` style properties of Button controls to point to these image files. Your Button controls reflect these new skins:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesTypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        overSkin: Embed("../assets/orb_over_skin.gif");
        upSkin: Embed("../assets/orb_up_skin.gif");
        downSkin: Embed("../assets/orb_down_skin.gif");
     }
  </mx:Style>
  <mx:Button id="b1" label="Click Me"/>
</mx:Application>
```

Although you apply skins as style properties, you should think of each skin as a class. If you set the value of the `upSkin` property of a Button control (for example, to "myimage.jpg"), you are really changing the reference from the current skin class to this new class. Flex wraps each embedded asset as an inner class inside the current document's class.

Skins are embedded in the final SWF file of the Flex application. If you change the graphics files that comprise one or more skins, you must recompile your Flex application for the changes to take effect.

You can define skin properties in the following ways:

- Cascading Style Sheets (CSS files). For more information, see "Using style sheets to apply skins" on page 809.
- Inline. For more information, see "Applying skins inline" on page 811.
- The `setStyle()` method. For more information, see "Using the setStyle() method" on page 812.

One drawback to embedding images as skins is that they can become distorted if you resize the component that has a skin. You can use a technique called scale-9 to create skins that do not become distorted when the component is resized. For information on the scale-9 technique, see "Using scale-9 formatting with embedded images" on page 1127.

You must *embed* graphical skin assets in the application's SWF file. The reason for this is that in order to determine a component's minimum and preferred sizes, skin assets must be present as soon as the component is created. If you reference external assets at run time, Flex does not have the sizing information and, therefore, cannot render the skins properly.

The following example shows how to embed external assets using the `@Embed` keyword inline:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Button id="b1"
     label="Click Me"
     overSkin="@Embed(source='../assets/orb_over_skin.gif')"
     upSkin="@Embed(source='../assets/orb_up_skin.gif')"
     downSkin="@Embed(source='../assets/orb_down_skin.gif')"
  />
</mx:Application>
```

Using style sheets to apply skins

You can embed skins as properties of a CSS file in the `<mx:Style>` tag or in an external style sheet, just as you would apply any style property, such as `color` or `fontSize`.

When applying skins with CSS, you can use type or class selectors so that you can apply skins to one component, or all components of the same type.

The following example defines skins on the Button type selector. In this example, all Buttons controls get the new skin definitions.

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesTypeSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        overSkin: Embed("../assets/orb_over_skin.gif");
        upSkin: Embed("../assets/orb_up_skin.gif");
        downSkin: Embed("../assets/orb_down_skin.gif");
     }
  </mx:Style>
  <mx:Button id="b1" label="Click Me"/>
</mx:Application>
```

You can also apply skins to a single instance of a component by defining a class selector. The following example applies the custom style to the second button only:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesClassSelector.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     .myButtonStyle {
        overSkin: Embed("../assets/orb_over_skin.gif");
        upSkin: Embed("../assets/orb_up_skin.gif");
        downSkin: Embed("../assets/orb_down_skin.gif");
     }
  </mx:Style>
  <mx:Button id="b1" label="Click Me"/>
  <mx:Button id="b2" label="Click Me" styleName="myButtonStyle"/>
</mx:Application>
```

In some cases, you want to reskin subcomponents. The following example reskins the vertical ScrollBar control that appears in List controls:

```
<?xml version="1.0"?>
<!-- skins/SubComponentSkins.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     [Bindable]
     private var theText:String = "Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum.";
  ]]></mx:Script>

  <mx:Style>
     .myScrollStyle {
       upArrowUpSkin: Embed("../assets/uparrow_up_skin.gif");
      downArrowUpSkin: Embed("../assets/downarrow_up_skin.gif");
     }
  </mx:Style>

  <mx:TextArea id="ta1"
     width="400"
     height="50"
     verticalScrollPolicy="on"
     verticalScrollBarStyleName="myScrollStyle"
     text="{theText}"
  />

</mx:Application>
```

By setting the value of the `verticalScrollBarStyleName` property in List type selector, all vertical ScrollBar controls in List components have a custom skin. The ScrollBar controls in other parts of the application do not have the custom skin.

As with all style sheets, you can define the skins in a separate CSS file and use the `source` attribute of the `<mx:Style>` tag to point to that file; for example:

```
<mx:Style source="../stylesheets/MySkins.css"/>
```

You can also load style sheets at run time by compiling them into a SWF file. You then use the StyleManager class's `loadStyleDeclarations()` method to load the CSS-based SWF file at run time, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/runtime/BasicApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
        import mx.styles.StyleManager;

        public function applyRuntimeStyleSheet():void {
            StyleManager.loadStyleDeclarations("assets/BasicStyles.swf")
        }
        ]]>
    </mx:Script>

    <mx:Label text="Click the button to load a new CSS-based SWF file"/>
    <mx:Button id="b1" label="Click Me" click="applyRuntimeStyleSheet()"/>

</mx:Application>
```

For more information, see "Loading style sheets at run time" on page 744.

Applying skins inline

To apply a graphical skin inline, you embed the skin by using the skin property of the control, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/EmbedImagesInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Button id="b1"
     label="Click Me"
     overSkin="@Embed(source='../assets/orb_over_skin.gif')"
     upSkin="@Embed(source='../assets/orb_up_skin.gif')"
     downSkin="@Embed(source='../assets/orb_down_skin.gif')"
  />
</mx:Application>
```

The location of the skin asset is relative to the location of the MXML file that embeds it.

When embedding inline, you use `@Embed` (with an at (@) sign prefix) rather than `Embed`, which you use in CSS files.

Using the setStyle() method

Skins are defined as style properties, therefore, you can access them with the `setStyle()` and `getStyle()` methods. This lets you change skins during run time, or dynamically define them, as long as you embed the graphical asset at compile time.

To embed an image so that you can use it with the `setStyle()` method, you use the `[Embed]` metadata tag and assign a reference to a variable. You can then use the `setStyle()` method to apply that image as a skin to a component.

The following example embeds three images and applies those images as skins to an instance of a Button control:

```
<?xml version="1.0"?>
<!-- skins/EmbedWithSetStyle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="init()">
  <mx:Script><![CDATA[
     [Embed("../assets/orb_over_skin.gif")]
     public var os:Class;

     [Embed("../assets/orb_down_skin.gif")]
     public var ds:Class;

     [Embed("../assets/orb_up_skin.gif")]
     public var us:Class;

     private function init():void {
        b1.setStyle("upSkin",us);
        b1.setStyle("overSkin",os);
        b1.setStyle("downSkin",ds);
     }
  ]]></mx:Script>

  <mx:Button label="Click Me" id="b1"/>

</mx:Application>
```

To apply skins to all instances of a control, you use the StyleManager, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/EmbedWithStyleManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="init()">
  <mx:Script><![CDATA[
     import mx.styles.StyleManager;

     [Embed("../assets/orb_over_skin.gif")]
     public var os:Class;

     [Embed("../assets/orb_down_skin.gif")]
     public var ds:Class;

     [Embed("../assets/orb_up_skin.gif")]
     public var us:Class;

     private function init():void {
        StyleManager.getStyleDeclaration("Button").setStyle("upSkin", us);
        StyleManager.getStyleDeclaration("Button").setStyle("overSkin",
os);
        StyleManager.getStyleDeclaration("Button").setStyle("downSkin",
ds);
     }
  ]]></mx:Script>

  <mx:Button label="Click Me" id="b1"/>

</mx:Application>
```

For more information on using the `setStyle()` method, see "Using the setStyle() and getStyle() methods" on page 737. For more information on using the StyleManager, see "Using the StyleManager class" on page 730.

Using SWF files as skins

You can use SWF files as skins. You can use the entire SWF file as a single skin, or you can use one or more symbols inside the SWF file as a skin. To embed an entire SWF file, you point to the location of the SWF file with the `source` property in the `Embed` statement, as follows:

```
<?xml version="1.0"?>
<!-- skins/EmbedSWFSource.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        upSkin: Embed(source="../assets/SubmitButtonUpSkin.swf");
     }
  </mx:Style>
  <mx:Button id="b1"/>
</mx:Application>
```

To import a symbol from a SWF file, you use the `symbol` property to specify the symbol name that you want to use in addition to pointing to the location of the SWF file with the `source` property. You must separate each attribute of the `Embed` statement with a comma.

The following example replaces the Button control's `upSkin` property with an entire SWF file, but the `overSkin` and `downSkin` properties with individual symbols from a second SWF file:

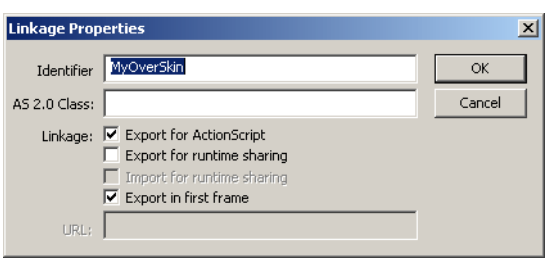
```
<?xml version="1.0"?>
<!-- skins/EmbedSymbolsCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        upSkin: Embed(source='../assets/SubmitButtonSkins.swf',
symbol='MyUpSkin');
        overSkin: Embed(source='../assets/SubmitButtonSkins.swf',
symbol='MyOverSkin');
        downSkin: Embed(source='../assets/SubmitButtonSkins.swf',
symbol='MyDownSkin');
     }
  </mx:Style>
  <mx:Button id="b1"/>
</mx:Application>
```

You use the same syntax when embedding skin symbols inline, as follows:

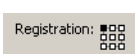
```
<?xml version="1.0"?>
<!-- skins/EmbedSymbolsInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Button id="b1"
     overSkin="@Embed(source='../assets/SubmitButtonSkins.swf',
symbol='MyOverSkin')"
     upSkin="@Embed(source='../assets/SubmitButtonSkins.swf',
symbol='MyUpSkin')"
     downSkin="@Embed(source='../assets/SubmitButtonSkins.swf',
symbol='MyDownSkin')"
  />
</mx:Application>
```

In the source FLA file, all symbols that you use must meet the following conditions:

- The symbol must be on the Stage. After you create an image file and convert it to a symbol, you must drag it from the library to the Stage. Flash does not export symbols that are not on the Stage. Alternatively, you can select the Export in First Frame option in the Linkage Properties dialog box.
- The symbol must have been exported for ActionScript with a linkage name. In Flash, you select the Export for ActionScript option in the symbol's Linkage Properties dialog box, as the following example shows:



  The linkage name is the name used by Flex. Symbol names are ignored.
- The FLA files cannot contain any ActionScript.
- The symbol must have an upper-left registration point that you select in the Convert to Symbol dialog box. The following example shows an upper-left registration point:



- You can use scale-9 image files (a grid with nine regions) so that the skin scales well when the component's size changes. You define the attributes for scale-9 images when you apply the skin in CSS, as the following example shows:

```
Button {
  overSkin: Embed(
```

```
        "../assets/orb_over_skin.gif",
        scaleGridTop=6,
        scaleGridLeft=12,
        scaleGridBottom=44,
        scaleGridRight=49
    );
}
```

For information on embedding assets that use the scale-9 technique, see "Using scale-9 formatting with embedded images" on page 1127.

Programmatic skinning

This section describes how to write programmatic skins as ActionScript classes, how to use the basic drawing methods of the Flash Graphics (flash.display.Graphics) package, and how to apply those skins to your Flex controls.

You can modify programmatic skins that come with Flex or create your own. For information on modifying existing skins, see "Resources for skins" on page 807. For information on using the programmatic skin recipe, see "Programmatic skins recipe" on page 818.

The simplest way to reskin a Flex component is to use an existing skin and apply that skin with CSS. You can use all the skins in the mx.skins.halo package and apply those classes to other controls. For example, you make your CheckBox appear like a RadioButton, you can apply the mx.skins.halo.RadioButton skin to the various states of the CheckBox control, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/UseHaloSkins.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     RadioButton {
        /*
           Defaults. Shown here for illustrative purposes.
        */
        disabledIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        downIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        overIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedDisabledIcon:
ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedDownIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedOverIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedUpIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        upIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
     }

     CheckBox {
        /*
           Applies the RadioButtonIcon skin to the CheckBox component's
states.
        */
        disabledIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        downIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        overIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedDownIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedDisabledIcon:
ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedOverIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        selectedUpIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
        upIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
     }
  </mx:Style>

  <mx:CheckBox id="cb1" label="Click Me"/>
  <mx:RadioButton id="rb1" label="Click Me"/>

</mx:Application>
```

To see all the states that a component uses a programmatic skin for, you can look at the component's entry in the defaults.css file.

When applying an existing programmatic skin to a component that does not by default use that skin class, you must be sure that the component supports the same states (such as downIcon and upIcon) as the skin's intended owner. Otherwise, you must create a custom programmatic skin class. For information on creating a custom programmatic skin, see "Programmatic skins recipe" on page 818.

When you create a custom programmatic skin, you must compile it; see "Compiling programmatic skins" on page 837.

After compiling a programmatic skin, you can apply the skin to components in your Flex applications. You can apply skins with CSS, use inline syntax, or use the `setStyle()` method. For more information, see "Applying programmatic skins" on page 843.

The programmatic skins used by Flex components are in the mx.skins.halo package. All of these skins extend one of the abstract base classes (ProgrammaticSkin, Border, or RectangularBorder). To change the look and feel of the Button control, for example, you can extend the abstract ProgrammaticSkin class and add logic for a Button control.

Using the techniques in this section, you can extend any skin in the default Halo theme, as well as the base skin classes.

Programmatic skins recipe

At a minimum, programmatic skin classes consist of a constructor, an `updateDisplayList()` method, and a set of getters and setters for the skin's properties. Programmatic skins generally extend one of the abstract classes in the mx.skins package.

To see examples of skins that follow the programmatic skin recipe, look at the concrete classes in the mx.skins.halo package. These are the skins that the Flex components use. Those skins follow the same recipe presented here.

The following example is a typical outline of a programmatic skin:

```
// skins/MySkinOutline.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;

  public class MySkinOutline extends ProgrammaticSkin {

     // Constructor.
     public function MySkinOutline():void {
        // Set default values here.
     }

     override protected function updateDisplayList(w:Number,
        h:Number):void {
        // Add styleable properties here.
        // Add logic to detect components state and set properties here.
        // Add drawing methods here.
     }
  }
} // Close unnamed package.
```

In your Flex application, you can apply a programmatic skin using the `ClassReference` statement in CSS:

```
<?xml version="1.0"?>
<!-- skins/ApplyMySkinOutline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        overSkin: ClassReference("MySkinOutline");
        upSkin: ClassReference("MySkinOutline");
        downSkin: ClassReference("MySkinOutline");
     }
  </mx:Style>
  <mx:Button id="b1" label="Click Me"/>
</mx:Application>
```

There are other methods you can use to apply programmatic skins in your Flex applications. For more information, see "Applying programmatic skins" on page 843.

Programmatic skins often define the appearance of the skin for multiple states of a control. For example, a button can have a different skin for the upSkin, downSkin, and overSkin states. These states refer to the default state, when the button is pressed, and when the mouse is over the button, respectively.

Use the following list of steps to create programmatic skins for your Flex controls. Each step is explained in more detail in the following sections.

To create programmatic skins for Flex controls:

1. Select one of the following abstract classes as a superclass for your programmatic skin:
   - ProgrammaticSkin
   - Border
   - RectangularBorder

   You can also extend one of the concrete classes in the mx.skins.Halo package. For more information, see "Selecting a superclass" on page 820.
2. Implement the `updateDisplayList()` method. Put all the drawing and styling calls in this method.

   For more information, see "Implementing the updateDisplayList() method" on page 821.
3. (Optional) Implement getters for the `measuredWidth` and `measuredHeight` properties.

   For more information, see "Implementing measuredWidth and measuredHeight getters" on page 825.
4. If the skin is a subclass of Border or RectangularBorder, implement a getter for the `borderMetrics` property.

   For more information, see "Implementing a getter for the borderMetrics property" on page 827.
5. Make properties styleable.

   If you create a skin that has properties that you want users to be able to set with CSS or with calls to the `setStyle()` method, you must add code to your skin class. For more information, see "Making properties styleable" on page 829.

When working in the skin class, it is possible to get a reference to the parent of the programmatic skin. You can use this reference to access properties of the parent component or call methods on it. For more information, see "Accessing the parent component" on page 834.

Selecting a superclass

The first step in creating a programmatic skin is to select a superclass for the skin.

You can extend the abstract base classes in the mx.skins package or the concrete classes in the mx.skins.halo package. Extending the former gives you greater control over the look and feel of your skins. Extending the latter is a good approach if you use the default behavior of the Flex components, but also want to add extra styling to them.

This section discusses extending one of the abstract base skin classes in the mx.skins package. You can apply the techniques described here to the concrete classes in the mx.skins.halo package.

Most skins extend the mx.skins.ProgrammaticSkin class, but you can select any one of the following as a superclass for your skin:

- The ProgrammaticSkin class implements the IFlexDisplayObject, ILayoutManagerClient, IInvalidating, and ISimpleStyleClient interfaces, so it is the easiest and most common superclass to use.
- The Border class extends the ProgrammaticSkin class and adds support for the `borderMetrics` property. Use this class or the RectangularBorder class if your skin defines the component's border.
- The RectangularBorder class extends the Border class and adds support for the `backgroundImage` style.

A programmatic skin class must implement the IFlexDisplayObject, IInvalidating, and ILayoutManagerClient interfaces. If the skin's appearance depends on the values of CSS styles, it must also implement the ISimpleStyleClient interface. The skin's superclass can be any subclass of the DisplayObject class (including the BitmapAsset class), as long as you implement those interfaces.

The following example defines the MySkin class with ProgrammaticSkin as its superclass:

```
package {
  import mx.skins.ProgrammaticSkin;
  public class MySkin extends ProgrammaticSkin {
    ...
  }
}
```

Implementing the updateDisplayList() method

The `updateDisplayList()` method defines the look of the skin. It is called after the skin's construction to initially draw the skin, and then is subsequently called whenever the component is resized, restyled, moved, or is interacted with in some way.

You use the drawing methods to draw the programmatic skin in the `updateDisplayList()` method.

When you implement the `updateDisplayList()` method, you must do the following:

- Use the `override` keyword to override the superclass's implementation.
- Set the return type to `void`.
- Declare the method as `protected`.

The `updateDisplayList()` method takes the height and width of the component as arguments. You use the values of these arguments as the boundaries for the region in which you can draw. The method returns `void`.

You use methods of the Graphics class (such as the `lineTo()` and `drawRect()` methods) to render the skin. To ensure that the area is clear before adding the component's shapes, you should call the `clear()` method before drawing. This method erases the results of previous calls to the `updateDisplayList()` method and removes all the images that were created by using previous draw methods. It also resets any line style that was specified with the `lineStyle()` method.

To use the methods of the Graphics package, you must import the flash.display.Graphics class, and any other classes in the flash.display package that you use, such as GradientType or Font. The following example imports all classes in the flash.display package:

```
import flash.display.*;
```

The following example draws a rectangle as a border around the component with the `drawRect()` method:

```
g.drawRect(0, 0, width, height);
```

The following example draws an *X* with a border around it:

```
// skins/CheckboxSkin.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;

  public class CheckboxSkin extends ProgrammaticSkin {

     // Constructor.
     public function CheckboxSkin():void {
        // Set default values here.
     }

  override protected function updateDisplayList(w:Number, h:Number):void {
     var g:Graphics = graphics;
     g.clear();
     g.beginFill(0xFFFFFF,1.0);
     g.lineStyle(2, 0xFF0000);
     g.drawRect(0, 0, w, h);
     g.endFill();
     g.moveTo(0, 0);
     g.lineTo(w, h);
     g.moveTo(0, h);
     g.lineTo(w, 0);
  }
  }
} // Close unnamed package.
```

For a description of common methods of the Graphics package, see "Drawing programmatically" on page 837. For details about these methods, see *Adobe Flex 2 Language Reference*.

One common task performed in the `updateDisplayList()` method is to change properties of the skin, depending on the current state of the control. For example, if you define a programmatic skin for a Button control, you can change the border thickness or color of the background when the user moves the mouse over or clicks on the Button control.

You check the state by using the `name` property of the skin. The `name` is the current name of the skin. For example, if you define a programmatic skin for a Button control, the `name` property could be any of the skin states: `downSkin`, `upSkin`, `overSkin`, `disabledSkin`, `selectedDisabledSkin`, `selectedDownSkin`, `selectedOverSkin`, or `selectedUpSkin`.

The following example checks which state the Button control is in and adjusts the line thickness and background fill color appropriately. The result is that when the user clicks the Button control, Flex redraws the skin to change the line thickness to 2 points. When the user releases the mouse button, the skin redraws again and the line thickness returns to its default value of 4. The background fill color also changes depending on the Button control's state.

```
// skins/ButtonStatesSkin.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;

  public class ButtonStatesSkin extends ProgrammaticSkin {

     public var backgroundFillColor:Number;
     public var lineThickness:Number;

     // Constructor.
     public function ButtonStatesSkin():void {
        // Set default values.
        backgroundFillColor = 0xFFFFFF;
        lineThickness = 4;
     }

    override protected function updateDisplayList(w:Number, h:Number):void
{
        // Depending on the skin's current name, set values for this skin.
        switch (name) {
           case "upSkin":
            lineThickness = 4;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "overSkin":
            lineThickness = 4;
            backgroundFillColor = 0xCCCCCC;
            break;
           case "downSkin":
            lineThickness = 2;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "disabledSkin":
            lineThickness = 2;
            backgroundFillColor = 0xCCCCCC;
            break;
        }
```

```
        // Draw the box using the new values.
        var g:Graphics = graphics;
        g.clear();
        g.beginFill(backgroundFillColor,1.0);
        g.lineStyle(lineThickness, 0xFF0000);
        g.drawRect(0, 0, w, h);
        g.endFill();
        g.moveTo(0, 0);
        g.lineTo(w, h);
        g.moveTo(0, h);
        g.lineTo(w, 0);
    }
  }
} // Close unnamed package.
```

If you use a single programmatic skin class to define multiple states of a control, you must apply the skin to all appropriate states of that control in your Flex application; for example:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStatesSkin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     Button {
        overSkin: ClassReference("ButtonStatesSkin");
        upSkin: ClassReference("ButtonStatesSkin");
        downSkin: ClassReference("ButtonStatesSkin");
     }
  </mx:Style>
  <mx:Button id="b1" label="Click Me"/>
</mx:Application>
```

If the skin is a subclass of RectangularBorder, you must also call `super.updateDisplayList()` from within the body of the `updateDisplayList()` method.

Implementing measuredWidth and measuredHeight getters

The `measuredWidth` and `measuredHeight` properties define the default width and height of a component. You can implement getter methods for the `measuredWidth` and `measuredHeight` properties, but it is not required by most skins. Some skins such as the skins that define the ScrollBar arrows do require that you implement these getters. If you do implement these getters, you must specify the `override` keyword when implementing the superclass's getter methods, and you must make the getters public.

The `measuredWidth` and `measuredHeight` getters typically return a constant number. The Flex application usually honors the measured sizes, but not always. If these getters are omitted, the values of `measuredWidth` and `measuredHeight` are set to the default value of 0.

The following example sets the `measuredWidth` and `measuredHeight` properties to 10, and then overrides the getters:

```
// skins/ButtonStatesWithMeasuredSizesSkin.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;

  public class ButtonStatesWithMeasuredSizesSkin extends ProgrammaticSkin {

     public var backgroundFillColor:Number;
     public var lineThickness:Number;

     private var _measuredWidth:Number;
     private var _measuredHeight:Number;

     // Constructor.
     public function ButtonStatesWithMeasuredSizesSkin():void {
        // Set default values.
        backgroundFillColor = 0xFFFFFF;
        lineThickness = 4;

        _measuredHeight = 100;
        _measuredWidth = 150;
     }

     override public function get measuredWidth():Number {
        return _measuredWidth;
     }

     override public function get measuredHeight():Number {
        return _measuredHeight;
     }

    override protected function updateDisplayList(w:Number, h:Number):void
{
        // Depending on the skin's current name, set values for this skin.
        switch (name) {
           case "upSkin":
            lineThickness = 4;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "overSkin":
            lineThickness = 4;
            backgroundFillColor = 0xCCCCCC;
```

```
            break;
           case "downSkin":
            lineThickness = 2;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "disabledSkin":
            lineThickness = 2;
            backgroundFillColor = 0xCCCCCC;
            break;
        }

        // Draw the box using the new values.
        var g:Graphics = graphics;
        g.clear();
        g.beginFill(backgroundFillColor,1.0);
        g.lineStyle(lineThickness, 0xFF0000);
        g.drawRect(0, 0, w, h);
        g.endFill();
        g.moveTo(0, 0);
        g.lineTo(w, h);
        g.moveTo(0, h);
        g.lineTo(w, 0);
    }
  }
} // Close unnamed package.
```

Implementing a getter for the borderMetrics property

The `borderMetrics` property defines the thickness of the border on all four sides of a programmatic skin. If the programmatic skin is a subclass of Border or RectangularBorder, you must implement a getter for the `borderMetrics` property. Otherwise, this step is optional. This property is of type EdgeMetrics, so your getter must set EdgeMetrics as the return type.

The following example gets the `borderThickness` style and uses that value to define the width of the four sides of the border, as defined in the EdgeMetrics constructor:

```
// skins/ButtonStatesWithBorderMetricsSkin.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;
  import mx.core.EdgeMetrics;

  public class ButtonStatesWithBorderMetricsSkin extends ProgrammaticSkin {

     public var backgroundFillColor:Number;
     public var lineThickness:Number;

     private var _borderMetrics:EdgeMetrics;

     // Constructor.
     public function ButtonStatesWithBorderMetricsSkin():void {
        // Set default values.
        backgroundFillColor = 0xFFFFFF;
        lineThickness = 4;
     }

     public function get borderMetrics():EdgeMetrics {
        if (_borderMetrics) {
           return _borderMetrics;
        }
        var borderThickness:Number = getStyle("borderThickness");
        _borderMetrics = new EdgeMetrics(borderThickness, borderThickness,
borderThickness, borderThickness);
        return _borderMetrics;
     }

     override protected function updateDisplayList(w:Number, h:Number):void
{
        // Depending on the skin's current name, set values for this skin.
        switch (name) {
           case "upSkin":
            lineThickness = 4;
            backgroundFillColor = 0xFFFFFF;
            break;
           case "overSkin":
            lineThickness = 4;
            backgroundFillColor = 0xCCCCCC;
            break;
```

```
            case "downSkin":
             lineThickness = 2;
             backgroundFillColor = 0xFFFFFF;
             break;
            case "disabledSkin":
             lineThickness = 2;
             backgroundFillColor = 0xCCCCCC;
             break;
        }

        // Draw the box using the new values.
        var g:Graphics = graphics;
        g.clear();
        g.beginFill(backgroundFillColor,1.0);
        g.lineStyle(lineThickness, 0xFF0000);
        g.drawRect(0, 0, w, h);
        g.endFill();
        g.moveTo(0, 0);
        g.lineTo(w, h);
        g.moveTo(0, h);
        g.lineTo(w, 0);
    }
  }
} // Close unnamed package.
```

Making properties styleable

In many cases, you define a programmatic skin that defines style properties, such as the background color of the skin, the border thickness, or the roundness of the corners. You can make these properties styleable so that your users can change their values in a CSS file or with the `setStyle()` method from inside their Flex applications. You cannot set styles that are defined in programmatic skins by using inline syntax.

To make a custom property styleable, add a call to the `getStyle()` method in the `updateDisplayList()` method and specify that property as the method's argument. When Flex renders the skin, it calls `getStyle()` on that property to find a setting in CSS or on the display list. You can then use the value of the style property when drawing the skin.

You should wrap this call to the `getStyle()` method in a check to see if the style exists. If the property was not set, the result of the `getStyle()` method can be unpredictable.

The following example verifies if the property is defined before assigning it a value:

```
if (getStyle("lineThickness")) {
  _lineThickness = getStyle("lineThickness");
}
```

You must define a default value for the skin's styleable properties. You usually do this in the skin's constructor function. If you do not define a default value, the style property is set to `NaN` or `undefined` if the Flex application does not define that style. This can cause a run-time error.

The following example of the MyButtonSkin programmatic skin class defines default values for the `_lineThickness` and `_backgroundFillColor` styleable properties in the skin's constructor. It then adds calls to the `getStyle()` method in the `updateDisplayList()` method to make these properties styleable:

```
// skins/ButtonStylesSkin.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;

  public class ButtonStylesSkin extends ProgrammaticSkin {

     public var _backgroundFillColor:Number;
     public var _lineThickness:Number;

     // Constructor.
     public function ButtonStylesSkin():void {
        // Set default values.
        _backgroundFillColor = 0xFFFFFF;
        _lineThickness=2;
     }

    override protected function updateDisplayList(w:Number, h:Number):void
{
        if (getStyle("lineThickness")) {
           // Get value of lineThickness style property.
           _lineThickness = getStyle("lineThickness");
        }
        if (getStyle("backgroundFillColor")) {
           // Get value of backgroundFillColor style property.
           _backgroundFillColor = getStyle("backgroundFillColor");
        }

        // Draw the box using the new values.
        var g:Graphics = graphics;
        g.clear();
        g.beginFill(_backgroundFillColor,1.0);
        g.lineStyle(_lineThickness, 0xFF0000);
        g.drawRect(0, 0, w, h);
        g.endFill();
        g.moveTo(0, 0);
        g.lineTo(w, h);
        g.moveTo(0, h);
        g.lineTo(w, 0);
     }
```

```
  }
} // Close unnamed package.
```

In your Flex application, you can set the values of styleable properties by using CSS or the `setStyle()` method.

The following example sets the value of a styleable properties on all Button controls with a CSS:

```
<?xml version="1.0"?>
<!-- skins/ApplyButtonStylesSkin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Style>
     Button {
        upSkin:ClassReference('ButtonStylesSkin');
        downSkin:ClassReference('ButtonStylesSkin');
        overSkin:ClassReference('ButtonStylesSkin');
        disabledSkin:ClassReference('ButtonStylesSkin');
        lineThickness:4;
        backgroundFillColor:#CCCCCC;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     public function changeLineThickness(e:Event):void {
        var t:int = Number(b1.getStyle("lineThickness"));
        if (t == 4) {
           b1.setStyle("lineThickness",1);
        } else {
           b1.setStyle("lineThickness",4);
        }
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Change Line Thickness"
click="changeLineThickness(event)"/>

  <mx:Button id="b2"/>
</mx:Application>
```

When using the `setStyle()` method to set the value of a style property in your Flex application, you can set the value of a styleable property on a single component instance (as in the previous example) or on all instances of a component. The following example uses the `setStyle()` method to set the value of a styleable property on all instances of the control (in this case, all Button controls):

```
<?xml version="1.0"?>
<!-- skins/ApplyGlobalButtonStylesSkin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Style>
     Button {
        upSkin:ClassReference('ButtonStylesSkin');
        downSkin:ClassReference('ButtonStylesSkin');
        overSkin:ClassReference('ButtonStylesSkin');
        disabledSkin:ClassReference('ButtonStylesSkin');
        lineThickness:4;
        backgroundFillColor:#CCCCCC;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     public function changeLineThickness(e:Event):void {
        var t:int = Number(b1.getStyle("lineThickness"));
        if (t == 4) {

StyleManager.getStyleDeclaration("Button").setStyle("lineThickness", 1);
        } else {

StyleManager.getStyleDeclaration("Button").setStyle("lineThickness", 4);
        }
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Change Line Thickness"
click="changeLineThickness(event)"/>

</mx:Application>
```

If you do not set the values of these properties using either CSS or the `setStyle()` method, Flex uses the default values of the properties that you set in the skin's constructor.

To get the value of an existing style property, such as `color` or `fontSize`, you do not have to wrap the call to the `getStyle()` method in a check for the property's existence. This is because Flex creates a CSSStyleDeclaration that defines the default values of all of a component's styles. Pre-existing style properties are never undefined. Style properties that you add to a custom skin are not added to this CSSStyleDeclaration because the component does not know that the property is a style property.

Custom skin properties that you define as styleable are noninheritable. So, subclasses or children of that component do not inherit the value of that property.

Accessing the parent component

It is possible to get a reference to the parent of the programmatic skin from within the programmatic skin class. You can use this reference to access properties of the parent component or call methods on it.

You can access the parent from the `updateDisplayList()` method by using the skin's `parent` property. You cannot access the parent in the skin's constructor because the skin has not yet been added to the parent control. The value of the skin's `parent` property is set when the parent component calls the `addChild()` method to add the skin as a child.

When instantiating components with programmatic skins, the order of events is as follows:

1. Create an instance of the parent component.
2. Create an instance of the skin class.
3. Call the `addChild()` method on the parent component to add the skin class.

To get a reference to the skin's parent, you must cast the skin's `parent` property to a UIComponent. The skin inherits this read-only property from the IFlexDisplayObject interface. You should also confirm that the parent is a UIComponent by using the `is` operator, because Flex throws a run-time error if the cast cannot be made.

The following example gets the class name of the parent control and draws the border and fill, depending on the type of component the parent is:

```
package {

import flash.display.GradientType;
import flash.display.Graphics;
import mx.skins.Border;
import mx.styles.StyleManager;
import mx.utils.ColorUtil;
import mx.skins.halo.HaloColors;
import mx.core.UIComponent;

public class IconSkin extends Border {

  public function IconSkin() {
     //super();
  }

  override public function get measuredWidth():Number {
     return 14;
  }

  override public function get measuredHeight():Number {
     return 14;
  }

  override protected function updateDisplayList(w:Number, h:Number):void {
     super.updateDisplayList(w, h);

     // User-defined styles
     var borderColor:uint = getStyle("borderColor");
     var fillAlphas:Array = getStyle("fillAlphas");
     var fillColors:Array = getStyle("fillColors");
     StyleManager.getColorNames(fillColors);
     var highlightAlphas:Array = getStyle("highlightAlphas");
     var themeColor:uint = getStyle("themeColor");

     var r:Number = width / 2;

     var upFillColors:Array;
     var upFillAlphas:Array;

     var disFillColors:Array;
     var disFillAlphas:Array;

     var g:Graphics = graphics;
     g.clear();

     var myParent:String;
```

```
     switch (name) {
        case "upIcon": {
           upFillColors = [ fillColors[0], fillColors[1] ];
           upFillAlphas = [ fillAlphas[0], fillAlphas[1] ];

           if (parent is UIComponent) {
            myParent = String(UIComponent(parent).className);
           }
           if (myParent=="RadioButton") {
            // RadioButton border
           g.beginGradientFill(GradientType.LINEAR,[ borderColor, 0x000000
], [100,100], [0,0xFF], verticalGradientMatrix(0,0,w,h));
            g.drawCircle(r,r,r);
            g.drawCircle(r,r,(r-1));
            g.endFill();

            // RadioButton fill
            g.beginGradientFill(GradientType.LINEAR, upFillColors,
upFillAlphas, [0,0xFF], verticalGradientMatrix(1,1,w-2,h-2));
            g.drawCircle(r,r,(r-1));
            g.endFill();
           } else if (myParent=="CheckBox") {
            // CheckBox border
            drawRoundRect(0,0,w,h,0,[borderColor, 0x000000],1,
verticalGradientMatrix(0,0,w,h), GradientType.LINEAR, null, {x: 1,y:1,w:w-
2,h:h-2,r:0});

            // CheckBox fill
            drawRoundRect(1,1,w-2,h-2,0,upFillColors, upFillAlphas,
verticalGradientMatrix(1,1,w-2,h-2));
           }

           // top highlight
           drawRoundRect(1,1,w-2,(h-2)/2,{tl:r,tr:r,bl:0,br:0},[0xFFFFFF,
0xFFFFFF], highlightAlphas, verticalGradientMatrix(0,0,w-2,(h-2)/2));
        }

        // Insert other cases such as downIcon and overIcon here.

     }
  }
}

}
```

Compiling programmatic skins

When you compile an application that uses programmatic skins, you treat programmatic skins as you would treat any ActionScript class. You must add the skins to the `source-path` argument of the compiler. If the skins are in the same directory as the MXML file that you are compiling, you set the `source-path` to a period. The following example shows this with the mxmlc command-line compiler:

```
$ ./mxmlc -source-path=. c:/flex/MyApp.mxml
```

If the programmatic skins are not in a package, you must add them to the unnamed package to make them externally visible. Otherwise, mxmlc throws a compiler error. To do this, you surround the class with a package statement, as the following example shows:

```
package { // Open unnamed package.
  import flash.display.*;
  import mx.skins.ProgrammaticSkin;

  public class MySkin extends ProgrammaticSkin {
    ...
  }
} // Close unnamed package.
```

In your Flex application, you must import the appropriate classes or packages that define the programmatic skins in a script block or helper class.

Drawing programmatically

You use the drawing methods of the Graphics class to draw the parts of a programmatic skin. These methods let you describe fills or gradient fills, define line sizes and shapes, and draw lines. By combining these very simple drawing methods, you can create complex shapes that make up your component skins.

The following table briefly describes the most commonly used drawing methods in the Graphics package:

Method	Summary
`beginFill()`	Begins drawing a fill; for example: `beginFill(0xCCCCFF,1);` If an open path exists (that is, if the current drawing position does not equal the previous position that you specified in a `moveTo()` method) and it has a fill associated with it, that path is closed with a line, and then filled.
`beginGradientFill()`	Begins drawing a gradient fill. If an open path exists (that is, if the current drawing position does not equal the previous position that you specified in a `moveTo()` method), and it has a fill associated with it, that path is closed with a line, and then filled.
`clear()`	Removes all the drawing output associated with the current object. The `clear()` method takes no arguments.

`curveTo()`	Draws a curve using the current line style; for example:
	`moveTo(500, 500);`
	`curveTo(600, 500, 600, 400);`
	`curveTo(600, 300, 500, 300);`
	`curveTo(400, 300, 400, 400);`
	`curveTo(400, 500, 500, 500);`

`drawCircle()`	Draws a circle after you set the line style and fill. You pass the method the x and y position of the circle, as well as the radius, as the following example shows: `drawCircle(10,10,50);`
`drawRect()`	Draws a rectangle once you set the line style and fill. You pass the method the *x* and *y* positions of the rectangle, as well as the length and width of the rectangle, as the following example shows: `drawRect(10,10,100,20);`
`drawRoundRect()`	Draws a rectangle with rounded corners, after you set the line and fill. You pass the method the x and y position of the rectangle, length and height of the rectangle, and the width and height of the ellipse that is used to draw the rounded corners, as the following example shows: `drawRoundRect(10,10,100,20,9,5)`
`endFill()`	Ends the fill specified by `beginFill()` or `beginGradientFill()` methods. The `endFill()` method takes no arguments. If the current drawing position does not equal the previous position that you specified in a `moveTo()` method and a fill is defined, the path is closed with a line, and then filled.

Method	Summary
`lineStyle()`	Defines the stroke of lines created with subsequent calls to the `lineTo()` and `curveTo()` methods. The following example sets the line style to a 2-point gray line with 100% opacity: `lineStyle(2,0xCCCCCC,1)` You can call the `lineStyle()` method in the middle of drawing a path to specify different styles for different line segments within a path. Calls to the `clear()` method reset line styles back to `undefined`.

`lineTo()`	Draws a line using the current line style. The following example draws a triangle:
	`moveTo (200, 200);`
	`lineTo (300, 300);`
	`lineTo (100, 300);`
	`lineTo (200, 200);`
	If you call the `lineTo()` method before any calls to the `moveTo()` method, the current drawing position returns to the default value of (0, 0).

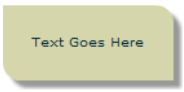
`moveTo()`	Moves the current drawing position to the specified coordinates; for example: `moveTo(100,10);`

The following example draws a triangle:

```
// skins/CheckBoxAsArrowSkin.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.skins.Border;
  import mx.skins.ProgrammaticSkin;
  import mx.styles.StyleManager;

  public class CheckBoxAsArrowSkin extends ProgrammaticSkin {

     // Constructor.
     public function CheckBoxAsArrowSkin():void {
        // Set default values.
     }

    override protected function updateDisplayList(w:Number, h:Number):void
{
        var unscaledHeight:Number = 2;
        var unscaledWidth:Number = 2;

        var arrowColor:Number;

        var g:Graphics = graphics;
        g.clear();

        switch (name) {
           case "upIcon":
           case "selectedUpIcon": {
            arrowColor = 0x666666;
            break;
           }
           case "overIcon":
           case "downIcon":
           case "selectedOverIcon":
           case "selectedDownIcon": {
            arrowColor = 0xCCCCCC;
            break;
           }
        }

        // Draw an arrow.
        graphics.lineStyle(1, 1, 1);
        graphics.beginFill(arrowColor);
        graphics.moveTo(unscaledWidth, unscaledHeight-20);
        graphics.lineTo(unscaledWidth-30, unscaledHeight+20);
        graphics.lineTo(unscaledWidth+30, unscaledHeight+20);
        graphics.lineTo(unscaledWidth, unscaledHeight-20);
```

```
            graphics.endFill();
        }
    }
} // Close unnamed package.
```

The ProgrammaticSkin class also defines drawing methods, the most common of which is the `drawRoundRect()` method. This method programmatically draws a rectangle and lets you set the corner radius, gradients, and other properties. You can use this method to customize borders of containers so that they might appear as following example shows:



The following code uses the `drawRoundRect()` method to draw this custom VBox border:

```
// skins/CustomContainerBorderSkin.as

package { // Use unnamed package if this skin is not in its own package.

  // Import necessary classes here.
  import flash.display.Graphics;
  import mx.graphics.RectangularDropShadow;
  import mx.skins.RectangularBorder;

  public class CustomContainerBorderSkin extends RectangularBorder {

     private var dropShadow:RectangularDropShadow;

     // Constructor.
     public function CustomContainerBorderSkin() {
     }

     override protected function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number):void {

        super.updateDisplayList(unscaledWidth, unscaledHeight);

        var cornerRadius:Number = getStyle("cornerRadius");
        var backgroundColor:int = getStyle("backgroundColor");
        var backgroundAlpha:Number = getStyle("backgroundAlpha");
        graphics.clear();

        // Background
        drawRoundRect(0, 0, unscaledWidth, unscaledHeight, {tl: 0,
tr:cornerRadius, bl: cornerRadius, br: 0}, backgroundColor,
backgroundAlpha);

        // Shadow
        if (!dropShadow)
           dropShadow = new RectangularDropShadow();

        dropShadow.distance = 8;
        dropShadow.angle = 45;
        dropShadow.color = 0;
        dropShadow.alpha = 0.4;
        dropShadow.tlRadius = 0;
        dropShadow.trRadius = cornerRadius;
        dropShadow.blRadius = cornerRadius;
        dropShadow.brRadius = 0;
        dropShadow.drawShadow(graphics, 0, 0, unscaledWidth,
unscaledHeight);
     }
  }
}
```

In your Flex application, you apply this skin as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/ApplyContainerBorderSkin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:VBox id="vb1"
     borderSkin="CustomContainerBorderSkin"
     backgroundColor="0xCCCC99"
     backgroundAlpha="0.8"
     cornerRadius="14"
     paddingLeft="20"
     paddingTop="20"
     paddingRight="20"
     paddingBottom="20"
  >
     <mx:Label text="Text Goes Here"/>
  </mx:VBox>

</mx:Application>
```

The "unscaled" width and height properties in the previous examples refer to the measurements of the skin as the skin itself understands them. These measurements ignore the fact that external components might have changed the dimensions of the skin. When working inside the component, it is best to use the unscaled measurements.

You can use any method to draw skins. For more information, see *Adobe Flex 2 Language Reference*.

Applying programmatic skins

As with graphical skins, you can apply programmatic skins to controls in your applications by using the following methods:

- CSS file; see "Using a CSS file" on page 844
- Inline; see "Using inline syntax" on page 845
- Using the `setStyle()` method and the StyleManager class; see "Using the setStyle() method and StyleManager with skins" on page 845

In each of these cases, you specify the programmatic skin's class, and not the full filename (for example, use "SampleSkin" instead of the full filename "SampleSkin.as") when you apply a programmatic skin to a component.

The programmatic skin's class must be in your source path when you compile the application. In addition, you must import the skin's class in an ActionScript block in your Flex application or helper class.

If your programmatic skins are in a named package, you separate the package levels with periods. For example, if the SampleButtonSkin class is in the my.skins package, you use my.skins.SampleButtonSkin when applying it, as you would reference any class in a package. If the programmatic skins are not in a named package, they must be in an unnamed package. You add them to an unnamed package by surrounding the class with a package statement.

For more information, see "Compiling programmatic skins" on page 837.

Using a CSS file

You apply programmatic skins in a CSS file by using the `ClassReference` directive. This directive takes a class name as the argument. The programmatic skin's class must be in your source path when you compile the application.

The following sample CSS file applies the SampleAccordionHeaderSkin and SampleButtonSkin programmatic skins to selected states of the AccordionHeader and Button controls:

```
/* assets/UseHaloSkins.css */
RadioButton {
    /*
        Defaults. Shown here for illustrative purposes.
    */
    disabledIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    downIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    overIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedDisabledIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedDownIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedOverIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedUpIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    upIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
}

CheckBox {
    /*
       Applies the RadioButtonIcon skin to the CheckBox component's states.
    */
    disabledIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    downIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    overIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedDownIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedDisabledIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedOverIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    selectedUpIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
    upIcon: ClassReference("mx.skins.halo.RadioButtonIcon");
}
```

As with any CSS file, you can define the style sheet with the `<mx:Style>` tag or reference an external style sheet, as the following example shows:

```
<?xml version="1.0"?>
<!-- skins/UseHaloSkinsStyleSheet.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style source="../assets/UseHaloSkins.css"/>

  <mx:CheckBox id="cb1" label="Click Me"/>
  <mx:RadioButton id="rb1" label="Click Me"/>

</mx:Application>
```

Using inline syntax

You can embed the programmatic skin class inline by specifying the class for each state that you reskin. You surround the class name with curly braces { }. The programmatic skin's class must be in your source path when you compile the application. If the class file is in the same directory as the Flex application, then you do not need to add it to your source path.

The following example applies the SampleButtonSkin programmatic skin to the Button control's `upSkin`, `overSkin`, `downSkin`, and `disabledSkin` states:

```
<?xml version="1.0"?>
<!-- skins/ApplyProgrammaticSkinsInline.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Button id="b1"
     label="Click Me"
     overSkin="{ButtonStatesSkin}"
     upSkin="{ButtonStatesSkin}"
     downSkin="{ButtonStatesSkin}"
     disabledSkin="{ButtonStatesSkin}"
  />
</mx:Application>
```

If the skin class is in a package, you must import the class into the application so that the compiler can resolve the class name, as the following example shows:

```
<mx:Script>
  import myPackage.*;
</mx:Script>
```

Using the setStyle() method and StyleManager with skins

You can apply a programmatic skin to a single instance of a control by using the `setStyle()` method. You specify the skin property to replace and the programmatic skin's class name in this method. The programmatic skin's class must be in your source path when you compile the application.

The following example applies the ButtonStatesSkin skin to the button1 control's upSkin state with the `setStyle()` method:

```
<?xml version="1.0"?>
<!-- skins/ApplyProgrammaticSkinsSetStyle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     public function changeSkins(e:Event):void {
        b1.setStyle("upSkin",ButtonStatesSkin);
        b1.setStyle("downSkin",ButtonStatesSkin);
        b1.setStyle("overSkin",ButtonStatesSkin);
        b1.setStyle("disabledSkin",ButtonStatesSkin);
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Change Skins" click="changeSkins(event)"/>

</mx:Application>
```

The reference to the ButtonStatesSkin class in the `setStyle()` method causes the compiler to link in the entire ButtonStatesSkin class at compile time. The resulting SWF file will be larger than if there were no reference to this class, even if the `changeSkins()` method is never called.

You can apply a programmatic skin to the states of all Button controls by using the StyleManager class. The following example applies the SampleButtonSkin skin to all Button controls' upSkin states:

```
<?xml version="1.0"?>
<!-- skins/ApplyProgrammaticSkinsStyleManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     public function changeSkins(e:Event):void {
        StyleManager.getStyleDeclaration("Button").setStyle("upSkin",
ButtonStatesSkin);
        StyleManager.getStyleDeclaration("Button").setStyle("downSkin",
ButtonStatesSkin);
        StyleManager.getStyleDeclaration("Button").setStyle("overSkin",
ButtonStatesSkin);
       StyleManager.getStyleDeclaration("Button").setStyle("disabledSkin",
ButtonStatesSkin);
     }
  ]]></mx:Script>

  <mx:Button id="b1" label="Change Skins" click="changeSkins(event)"/>

</mx:Application>
```

If the skin is in a package, you must import the class into the application so that the compiler can resolve the class name, as the following example shows:

```
<mx:Script>
  import myPackage.ButtonStatesSkin;
</mx:Script>
```

For more information on using the `setStyle()` method, see "Using the setStyle() and getStyle() methods" on page 737. For more information on using the StyleManager, see "Using the StyleManager class" on page 730.

Reskinning ToolTips

This section describes how to apply a programmatic skin to ToolTip controls. It illustrates some of the concepts presented earlier in this topic.

ToolTip skins are defined by the ToolTipBorder programmatic skin. This file is located in the mx.skins.halo package.

To reskin ToolTips, you edit the ToolTipBorder class file, and then apply the new skin to the ToolTip by using CSS. The following procedure describes the basic steps.

To reskin ToolTips by using CSS:

1. Open the mx.skins.halo.ToolTipBorder.as file. This file is included with the source files as described in "Resources for skins" on page 807.
2. Save the ToolTipBorder.as file under another name, and in a different location. The file name must be the same as the new class name.
3. Change the package from mx.skins.halo to your package name, or to the empty package, as the following example shows:

   ```
   //package mx.skins.halo { // Old package name
   package { // New, empty package
   ```

4. Change the class name in the file, as the following example shows:

   ```
   //public class ToolTipBorder extends RectangularBorder // Old name
   public class MyToolTipBorder extends RectangularBorder // New name
   ```

5. Change the constructor to reflect the new class name, as the following example shows:

   ```
   //public function ToolTipBorder() // Old constructor
   public function MyToolTipBorder() // New constructor
   ```

6. Comment out the versioning line, as the following example shows:

   ```
   //include "../../core/Version.as";
   ```

7. Edit the class file to change the appearance of the ToolTip. You do this by editing the "toolTip" case in the `updateDisplayList()` method. That is the method that draws the ToolTip's border and sets the default styles. Most commonly, you change the arguments of the `drawRoundRect()` method to change the appearance of the ToolTip.

   The following example adds a red tinge to the background of the ToolTip's box by replacing the default `backgroundColor` property with an array of colors:

   ```
   var highlightAlphas:Array = [0.3,0.0];
   drawRoundRect(3, 1, w-6, h-4, cornerRadius, [0xFF0000, 0xFFFFBB],
     backgroundAlpha);
   ```

8. Save the class file.

9. In your Flex application, edit the ToolTip's `borderSkin` style property to point to the new skin. You can do this in an `<mx:Style>` tag inside your application, use an external CSS file, or by using a custom theme file. The following example sets the `borderSkin` property to the new skin class:

   ```
   <?xml version="1.0"?>
   <!-- skins/ApplyCustomToolTipSkin.mxml -->
   <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
     <mx:Style>
        ToolTip {
           borderSkin: ClassReference("MyToolTipBorder");
        }
     </mx:Style>

     <mx:Button id="b1" label="Click Me" toolTip="Click this button"/>
   </mx:Application>
   ```

   You must specify the full package name in the CSS reference. In this example, the file MyToolTipBorder.as is in an empty package and, therefore, has no package designation such as mx.skins.halo.

10. Compile and run the application with the new skin file in the source path. You do this by setting the value of the compiler's `source-path` option.

    If the skin class is in the same directory as the application, you do not have to add its location to the source path. The current directory is assumed. For more information, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Creating themes

A *theme* is a collection of style definitions and skins that define the look and feel of a Flex application. Theme files can include both graphical and programmatic skins, as well as style sheets.

A theme takes the form of a SWC file that can be applied to a Flex application. You compile a SWC file using the compc command-line compiler utility.

By compiling a SWC file and then using that SWC file as a theme in your application, you remove the burden of compiling all the skin files when you compile your main application. This can make compilation faster and make problems with the application code easier to debug. In addition, a SWC file is easier to transplant onto another application than are a set of classes and image files.

For more information on creating and applying themes, see "About themes" on page 756.

CHAPTER 21

Using Item Renderers and Item Editors

21

You can customize the appearance and behavior of cells in list-based data provider controls that you can use in Adobe Flex applications, including the DataGrid, HorizontalList, List, Menu, MenuBar, TileList, and Tree controls. To control the appearance of cells in list controls, you create custom item renderers and item editors.

This topic introduces item renderers and item editors, and describes the different ways that you can create them in your Flex applications. The next topic, Chapter 22, "Working with Item Editors," on page 903, contains additional information for creating more advanced item editors.

Contents

About item renderers

Flex supports several controls that you can use to represent lists of items. These controls let the application user scroll through the item list and select one or more items from the list. All Flex list components are derived from the ListBase class, and include the following controls:

- DataGrid
- HorizontalList
- List
- Menu
- MenuBar
- TileList

- Tree

A list control gets its data from a *data provider*, which is a collection of objects. For example, a Tree control reads data from a data provider to define the structure of the tree and any associated data that is assigned to each tree node.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same data provider, switch data providers for a component at run time, and modify the data provider so that changes are reflected by all components that use the data provider.

You can think of the data provider as the model, and the Flex components as the view of the model. By separating the model from the view, you can change one without changing the other.

Each list control has a default mechanism for controlling the display of data, or *view*, and lets you override that default. To override the default view, you create a custom *item renderer*.

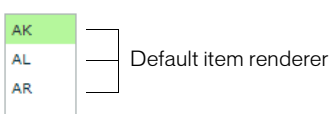
In addition to controlling the display of data using an item renderer, the DataGrid, List, and Tree controls let users edit the data. For example, you could let users update the Quantity column of a DataGrid control if you are using it for a shopping cart. As with the display of data, the list controls have a default mechanism for letting users edit data that you can override by creating a custom *item editor*.

Default item rendering and cell editing

Each list control has a default item renderer defined for it. The simplest item renderer is the DataGridItemRenderer class, which defines the item renderer for the DataGrid control. This item renderer assumes that each element in a data provider is a text string.

Other item renderers include the ListItemRenderer, MenuItemRenderer, MenuBarItem, TileListItemRenderer, and TreeItemRenderer. By default, these item renderers combine an image with text.

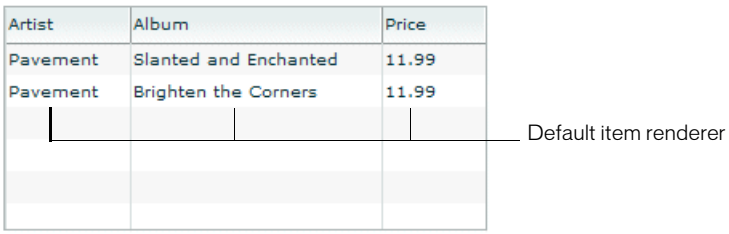
For example, the following image shows a List control that displays three items using its default item renderer:



In this example, the List control uses the default item renderer to display each of the three strings that represent postal codes for Alaska, Alabama, and Arkansas. You use the following code to create this List control:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\ListDefaultRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:List width="50" height ="75">
        <mx:dataProvider>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
            <mx:String>AR</mx:String>
        </mx:dataProvider>
    </mx:List>
</mx:Application>
```

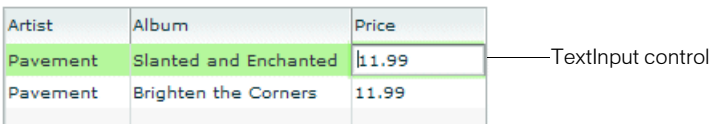
In the next example, the data provider for the DataGrid control contains fields for an artist, album name, and price. Each of these fields appears in the DataGrid control using the default item renderer, which displays data as text:



The following code creates the example shown in the previous image:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\DGDefaultRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
          import mx.collections.ArrayCollection;

          [Bindable]
          private var initDG:ArrayCollection = new ArrayCollection([
           {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
           {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}
          ]);
        ]]>
    </mx:Script>

    <mx:Panel paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10">

        <mx:DataGrid id="myGrid" dataProvider="{initDG}"
            width="100%" editable="true">
            <mx:columns>
                <mx:DataGridColumn dataField="Artist" resizable="true"/>
                <mx:DataGridColumn dataField="Album" resizable="true"/>
                <mx:DataGridColumn dataField="Price" resizable="true"/>
            </mx:columns>
        </mx:DataGrid>
    </mx:Panel>
</mx:Application>
```

An editable component lets the user modify the data in the list control, and propagates the changes in the data back to the underlying data provider of the control. The DataGrid, List, and Tree controls have a property named `editable` that, if set to `true`, lets you edit the contents of a cell. By default, the list controls use a TextInput control as the item editor. That means when you select a cell in the list control, Flex opens a TextInput control that contains the current contents of the cell and lets you edit it, as the following image shows:



In this image, you use the default item editor to edit the price of the first album in the DataGrid control.

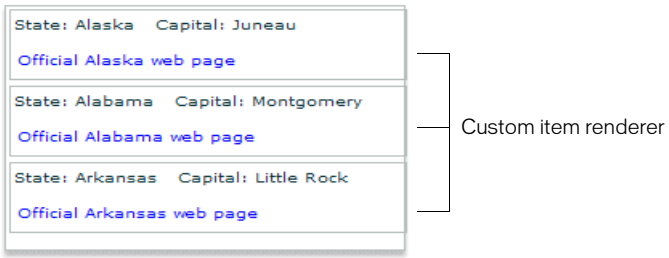
For more information on item editors, see Chapter 22, "Working with Item Editors," on page 903.

Using custom item renderers and item editors

To control the display of a list component, you write a custom item renderer or custom item editor. Your custom item renderers and item editors still use the underlying functionality of the list control, but let you control the display and editing of the data. Custom item renderers and item editors provide you with several advantages:

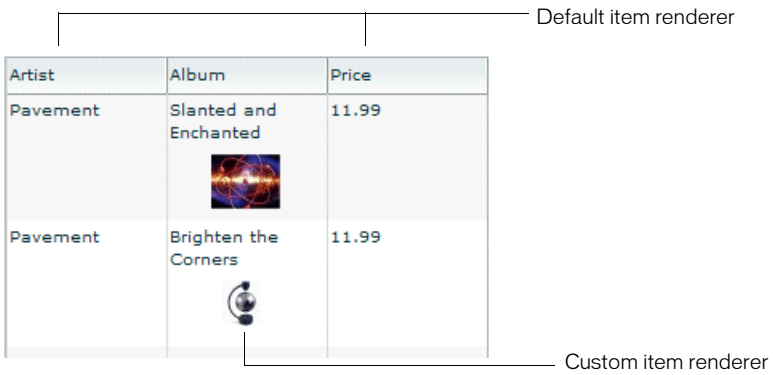
- You can create a more compelling user interface by replacing the display of text with a more user-intuitive appearance.
- You can combine multiple elements in a single list item, such as a label and an image.
- You can programmatically control the display of the data.

The following List control is a modification of the List control in the section "Default item rendering and cell editing" on page 852. In this example, you use a custom item renderer to display the state name, capital, and a URL to the state's official website in each list item:



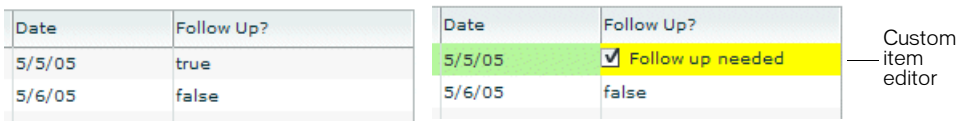
For the code for this example, see "Example: Using an item renderer with a List control" on page 897.

In the next image, you use a default item renderer for the first and third columns of the DataGrid control. You use a custom item renderer for the second column to display the album cover along with the album title in the DataGrid control.



For the code for this example, see "Creating a complex inline item renderer or item editor" on page 876.

Just as you can define a custom item renderer to control the display of a cell, you can use a custom item editor to edit the contents of the cell. For example, if the custom item renderer displays an image, you could define a custom item editor that uses a ComboBox control that lets users select an image from a list of available images. Or, you could use a CheckBox control to let the user set a `true` or `false` value for a cell, as the following example shows:



DataGrid control rendered normally

DataGrid control with custom item editor

For the code for this example, see "Creating a simple item editor component" on page 887.

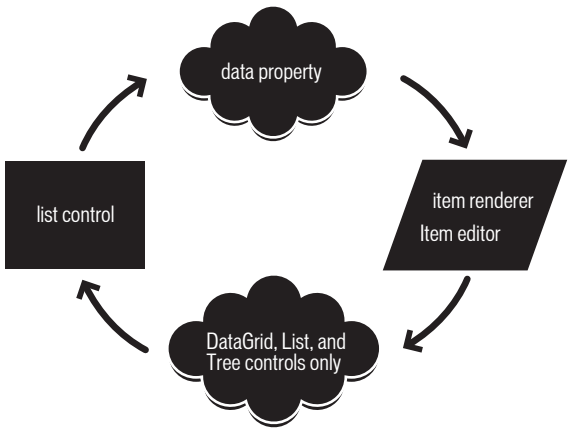
Using an item renderer does not imply that the control also has an item editor. Often, you do not allow your controls to be edited; that is, they are for display only.

You can also use an item editor without a corresponding item renderer. For example, you could display information such as a male/female or true/false as text using the default item renderer. But, you could then use a custom item editor with a ComboBox control that provides the user only a limited set of options to enter into the cell when changing the value.

Item renderer and item editor architecture

In order for an item renderer or item editor to work with the contents of a list control, the list control must be able to pass information to the item renderer or item editor. An item editor must also be able to pass updated information back to the list control.

The following image shows the relationship of a list control to an item renderer or item editor:



To pass information to an item renderer or item editor, Flex sets the `data` property of the item renderer or item editor to the cell data. In the previous image, you can see the `data` property that is used to pass the data to the item renderer or item editor.

By default, an item editor can pass back a single value to the list control that becomes the new value of the edited cell. To return more than one value, you must write additional code to return the data back to the List control. For more information, see "Example: Passing multiple values back from an item editor" on page 925.

Any Flex component that you want to use in an item renderer or item editor, and that requires access to the cell data of the list control, must implement the IDataRenderer interface to support the `data` property. You can use additional components in an item renderer that do not support the `data` property, if those components do not need to access data passed from the list control.

The contents of the `data` property depend on the type of control, as the following table shows:

Control	Contents of the data property
DataGrid	Contains the data provider element for the entire row of the DataGrid control. That means if you use a different item renderer for each cell of the DataGrid control, each cell in a row receives the same information in the `data` property.
Tree List	Contains the data provider element for the node of the List or Tree control.
HorizontalList TileList	Contains the data provider element for the cell.

Control	Contents of the data property
Menu	Contains the data provider element for the menu item.
MenuBar	Contains the data provider element for the menu bar item.

For more information on item editors, see Chapter 22, "Working with Item Editors," on page 903.

About item renderer and item editor interfaces

The Flex item renderer and item editor architecture is defined by several interfaces. Flex components that you can use as item renderers and item editors implement one or more of these interfaces. If you create your own custom component to use as an item renderer or item editor, your component must implement one or more of these interfaces, depending on how you intend to use it.

The following table describes the interfaces that you use with item renderers and item editors:

Interface	Use
IDataRenderer	Defines the `data` property used to pass information to an item renderer or item editor. All item renderers and item editors must implement this interface. Many Flex components implement this interface, such as the chart renderer classes, and many Flex controls.

Interface	Use
IDropInListItemRenderer	Defines the `listData` property required by drop-in item renderers and item editors. All drop-in item renderers and item editors must implement this interface. For more information, see "Creating drop-in item renderers and item editors" on page 870. The `listData` property is of type BaseListData, where the BaseListData class has three subclasses: DataGridListData, ListData, TreeListData. The actual data type of the value of the `listData` property depends on the control using the drop-in item renderer or item editor. For a DataGrid control, the value is of type DataGridListData, for a List control the value is of type ListData, and for a Tree control, the value is of type TreeListData.
IListItemRenderer	Defines the complete set of interfaces that an item renderer or item editor must implement to be used with any Flex control other than a Chart control. A renderer for a Chart control has to implement only the IDataRenderer interface. The set of interfaces includes the following: IDataRenderer, IFlexDisplayObject, ILayoutClient, IStyleable, IUIComponent. The UIComponent class implements all of these interfaces, except the IDataRenderer interface. Therefore, if you create a custom item renderer or item editor as a subclass of the UIComponent class, you have to implement only the IDataRenderer interface. If you implement a drop-in item renderer or item editor as a subclass of the UIComponent class, you have to implement only the IDataRenderer and IDropInListItemRenderer interfaces.

Application layout with item renderers and item editors

All list controls, with the exception of the TileList and HorizontalList controls, ignore item renderers when calculating the default height and width of their contents. Therefore, if you define a custom item renderer that requires a size other than its default, you should explicitly size the control.

Vertical controls such as the List and Tree controls use the `rowHeight` property to determine the default cell height. You can also set the `variableRowHeight` property to `true` if each row has a different height. Otherwise, Flex sets the height of each cell to 20 pixels.

For the HorizontalList control, Flex can use the value of the `columnWidth` property, if specified. For the TileList control, Flex uses the values of the `columnWidth` and `rowHeight` properties. If you omit these properties, Flex examines the control's data provider to determine the default size of each cell, including an item renderer, if specified. Otherwise, Flex sets the height and width of each cell to 50 pixels.

Creating an item renderer and item editor

One of the first decisions that you must make when using custom item renderers and item editors is to decide how to implement them. Flex lets you define item renderers and item editors in three different ways:

Drop-in item renderers and item editors Insert a single component to define an item renderer or item editor. For more information, see "Using a drop-in item renderer or item editor" on page 860.

Inline item renderers and item editors Define one or more components using child tags of the list control to define an item renderer or item editor. For more information, see "Using an inline item renderer or item editor" on page 862.

item renderer and item editor components Define an item renderer or item editor as a reusable component. For more information, see "Using a component as an item renderer or item editor" on page 863.

The following sections describe each technique.

Using a drop-in item renderer or item editor

For simple item renderers and item editors, such as using a NumericStepper control to edit a field of a DataGrid control, you can use a drop-in item editor. A *drop-in item renderer* or *drop-in item editor* is a Flex component that you specify as the value of the `itemRenderer` or `itemEditor` property of a list control.

In the following example, you specify the NumericStepper control as the item editor for a column of the DataGrid control.

```
<?xml version="1.0"?>
<!-- itemRenderers\dropin\DropInNumStepper.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1"
                headerText="Order #" />
            <mx:DataGridColumn dataField="quant"
                itemEditor="mx.controls.NumericStepper"
                editorDataField="value" />
        </mx:columns >
    </mx:DataGrid>
</mx:Application>
```

In this example, when the user selects a cell of the column to edit it, Flex displays the NumericStepper control in that cell. Flex automatically uses the `data` property to populate the NumericStepper control with the current value of the column.

You use the `editorDataField` property to specify the property of the item editor that returns the new data for the cell. By default, the list control uses the `text` field of the TextInput control to supply the new value; therefore, the default value of the `editorDataField` property is `"text"`. In this example, you specify that the `value` field of the NumericStepper supplies the new value for the cell.

You can only use a subset of the Flex components as drop-in item renderers and item editors; those components that implement the IDropInListItemRenderer interface. For more information on using drop-in item renderers and item editors, and for a list of controls that support drop-in item renderers and item editors, see "Creating drop-in item renderers and item editors" on page 870.

Using an inline item renderer or item editor

In the section "Using a drop-in item renderer or item editor" on page 860, the example shows how easy it is to use drop-in item renderers and item editors. The only drawback to using them is that you cannot configure them. You can only specify the drop-in item renderers and item editors as the values of a list control property.

To create more flexible item renderers and item editors, you develop your item renderer or item editor as an inline component. In the next example, you modify the previous example to use a NumericStepper control as an inline item editor. With an inline item editor, you can configure the NumericStepper control just as if you used it as a stand-alone control:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineNumStepper.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1" headerText="Order #"/>
            <mx:DataGridColumn dataField="quant" editorDataField="value">
                <mx:itemEditor>
                    <mx:Component>
                        <mx:NumericStepper stepSize="1" maximum="50"/>
                    </mx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, you define the NumericStepper control as the item editor for the column, and specify a maximum value of 50 and a step size of 1 for the NumericStepper control.

For more information on creating inline item renderers and item editors, see"Creating inline item renderers and editors" on page 875.

Using a component as an item renderer or item editor

One disadvantage of using drop-in and inline item renderers and editors is that you define the item renderer or editor in the application file; therefore, it is not reusable in another location in the application, or in another application. You can create a reusable item renderer or item editor as a Flex component, and then use that component anywhere in an application that requires the item renderer.

For example, the following code uses a NumericStepper control to define a custom item editor as an MXML component:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\myComponents\NSEditor.mxml -->
<mx:NumericStepper xmlns:mx="http://www.adobe.com/2006/mxml"
    stepSize="1"
    maximum="50"/>
```

The custom MXML component defines the item editor as a NumericStepper control. In this example, the custom item editor is named NSEditor and is implemented as an MXML component in the NSEditor.mxml file. You place the file NSEditor.mxml in the myComponents directory beneath your main application directory.

You can then use this component anywhere in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\MainNSEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myDG" dataProvider="{myDP}"
            variableRowHeight="true"
            editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1"
                headerText="Order #"/>
            <mx:DataGridColumn dataField="quant"
                itemEditor="myComponents.NSEditor"
                editorDataField="value"/>
        </mx:columns >
    </mx:DataGrid>
</mx:Application>
```

When the user selects a cell in the quant column of the DataGrid control, Flex displays a NumericStepper control with the current cell value.

You might have several locations in your application where users can modify a numeric value. You defined this item editor as a custom component; therefore, you can reuse it anywhere in your application.

For more information on creating item renderers and item editors as components, see "Creating item renderers and item editor components" on page 883.

Using editable controls in an item renderer

Item renderers do not impose restrictions on the types of Flex components that you can use in them. For example, you can use controls, such as the Label, LinkButton, Button, and Text controls to display data, but these controls do not let the user modify the contents of the control.

Or, you can use controls such as the CheckBox, ComboBox, and TextInput controls that both display data and let users interact with the control to modify or change it. For example, you could use a CheckBox control to display a selected (`true` value) or unselected (`false` value) cell in a DataGrid control.

When the user selects the cell of the DataGrid control that contains the CheckBox control, the user can interact with the control to change its state. To the user, it appears that the DataGrid control is editable.

However, an item renderer by default is not connected to the editing mechanism of the list control; it does not propagate changes to the list control's data provider, nor does it dispatch an event when the user modifies the cell. Although the list control appears to the user to be editable, it really is not.

In another example, the user changes the value of the CheckBox control, then sorts the DataGrid column. But, the DataGrid sorts the cell by the value in the data provider, not the value in the CheckBox control, so the user perceives that the sort works incorrectly.

You can manage this situation in several ways, including the following:

- Do not use controls in an item renderer that let users modify them (CheckBox, ComboBox, and others).
- Create custom versions of these controls to prohibit user interaction with them.
- Use the `rendererIsEditor` property of the list control to specify that the item renderer is also an item editor. For more information, see "Example: Using an item renderer as an item editor" on page 931.
- Write your own code for the item renderer and hosting control to pass data back from the item renderer when you do let users interact with it. For more information and an example, see "Example: Passing multiple values back from an item editor" on page 925.

Setting the itemRenderer or itemEditor property in ActionScript

The `itemRenderer` and `itemEditor` properties are of type IFactory. When you set these properties in MXML, the MXML compiler automatically casts the property value to the type ClassFactory, a class that implements the IFactory interface.

When you sets these properties in ActionScript, you must explicitly cast the property value to ClassFactory, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\AppListStateRendererEditorAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.core.ClassFactory;

            // Cast the value of the itemRenderer property
            // to ClassFactory.
            public function initCellEditor():void {
                myList.itemRenderer=new ClassFactory(RendererState);
            }
        ]]>
    </mx:Script>

    <mx:List id="myList" variableRowHeight="true"
            height="180" width="250"
            backgroundColor="white"
            initialize="initCellEditor();">
        <mx:dataProvider>
            <mx:Object label="Alaska"
                data="Juneau"
                webPage="http://www.state.ak.us/"/>
            <mx:Object label="Alabama"
                data="Montgomery"
                webPage="http://www.alabama.gov/" />
            <mx:Object label="Arkansas"
                data="Little Rock"
                webPage="http://www.state.ar.us/"/>
        </mx:dataProvider>
    </mx:List>
</mx:Application>
```

About the item renderer and item editor life cycle

Flex creates instances of item renderers and item editors as needed by your application for display and measurement purposes. Therefore, the number of instances of an item renderer or item editor in your application is not necessarily directly related to the number of visible item renderers. Also, if the list control is scrolled or resized, a single item renderer instance may be reused to represent more than one data item. Thus, you should not make assumptions about how many instances of your item renderer and item editor are active at any time.

Because Flex can reuse an item renderer, ensure that you fully define its state. For example, you use a CheckBox control in an item renderer to display a `true` (checked) or `false` (unchecked) value based on the current value of the `data` property. A common mistake is to assume that the CheckBox control in the item renderer is always in its default state of unchecked. Developers then write an item renderer to inspect the `data` property for a value of `true` and set the check of a CheckBox control if found.

However, you must take into account that the CheckBox may already be checked. So, you must inspect the `data` property for a value of `false`, and explicitly uncheck the control if it is checked.

Accessing the listData property

If a component implements the IDropInListItemRenderer interface, you can use its `listData` property to obtain information about the data passed to the component when you use the component in an item renderer or item editor. The `listData` property is of type BaseListData, where the BaseListData class defines the following properties:

Property	Description
`owner`	A reference to the list control that uses the item renderer or item editor.
`rowIndex`	The index of the row of the DataGrid, List, or Tree control relative to the currently visible rows of the control, where the first row is at an index of 1.
`text`	The text representation of the item data based on the List class's `itemToLabel()` method.

The BaseListData class has three subclasses: DataGridListData, ListData, TreeListData that define additional properties. For example, the DataGridListData class adds the `columnIndex` and `dataField` properties that you can access from an item renderer or item editor.

The data type of the value of the `listData` property depends on the control that uses the item renderer or item editor. For a DataGrid control, the value is of type DataGridListData; for a List control, the value is of type ListData; and for a Tree control, the value is of type TreeListData.

The TextArea control is one of the Flex controls that implements the IDropInListItemRenderer interface The item renderer in the following example uses the `listData` property of the TextArea control to display the row and column of each item renderer in the DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\RendererDGListData.mxml -->
<mx:TextArea xmlns:mx="http://www.adobe.com/2006/mxml"
    preinitialize="initTA();">

    <mx:Script>
        <![CDATA[

            import mx.controls.dataGridClasses.DataGridListData;
            import flash.events.Event;

            public function initTA():void {
                addEventListener("dataChange", handleDataChanged);
            }

            public function handleDataChanged(event:Event):void {
                // Cast listData to DataGridListData.
                var myListData:DataGridListData =
                    DataGridListData(listData);

                // Access information about the data passed
                // to the cell renderer.
                text="row index: " + String(myListData.rowIndex) +
                    " column index: " + String(myListData.columnIndex);
            }
        ]]>
    </mx:Script>
</mx:TextArea>
```

Because you use this item renderer in a DataGrid control, the data type of the value of the `listData` property is DataGridListData. You use the `dataChange` event in this example to set the contents of the TextArea control every time the `data` property changes.

The DataGrid control in the following example uses this item renderer:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainDGListDataRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var initDG:ArrayCollection = new ArrayCollection([
                { Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', Date: '5/5/05'},
                { Company: 'Allied', Contact: 'Jane Smith',
                    Phone: '617-555-3434', Date: '5/6/05'}
            ]);
        ]]>
    </mx:Script>

    <mx:Panel paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10" >

        <mx:DataGrid id="myGrid" dataProvider="{initDG}"
                variableRowHeight="true">
            <mx:columns>
                <mx:DataGridColumn dataField="Company"
                    itemRenderer="myComponents.RendererDGListData"/>
                <mx:DataGridColumn dataField="Contact"
                    itemRenderer="myComponents.RendererDGListData"/>
                <mx:DataGridColumn dataField="Phone"
                    itemRenderer="myComponents.RendererDGListData"/>
                <mx:DataGridColumn dataField="Date"
                    itemRenderer="myComponents.RendererDGListData"/>
            </mx:columns>
        </mx:DataGrid>
    </mx:Panel>
</mx:Application>
```

This code produces the following image of the DataGrid control:

Company	Contact	Phone	Date
row index: 1 column index: 0	row index: 1 column index: 1	row index: 1 column index: 2	row index: 1 column index: 3
row index: 2 column index: 0	row index: 2 column index: 1	row index: 2 column index: 2	row index: 2 column index: 3

As you can see in this image, each cell of the DataGrid control displays its column and row index.

Creating drop-in item renderers and item editors

Several Flex controls are designed to work as item renderers and item editors. This lets you specify these controls as values of the `itemRenderer` or `itemEditor` property. When you specify one of these controls as a property value, it is called a drop-in item renderer or drop-in item editor.

To use a component as a drop-in item renderer or drop-in item editor, a component must implement the IDropInListItemRenderer interface. The following controls implement the IDropInListItemRenderer interface, making them usable directly as a drop-in item renderer or drop-in item editor:

- Button
- CheckBox
- DateField
- Image
- Label
- NumericStepper
- Text
- TextArea
- TextInput

You can define your own components for use as drop-in item renderers or drop-in item editors. The only requirement is that they, too, implement the IDropInListItemRenderer interface.

Using drop-in item renderers and item editors

When you use a control as a drop-in item renderer, Flex sets the control's default property to the current value of the cell. When you use a control as an item editor, the initial value of the cell becomes the current value of the control. Any edits made to the cell are copied back to the data provider of the list control.

The following table lists the components that you can use as drop-in item renderers and item editors, and the default property of the component:

Control	Default property	Notes
Button	`selected`	Cannot specify a label for the Button control.
CheckBox	`selected`	If used as a drop-in item renderer in a Tree control, the Tree displays only check boxes with no text.
DateField	`selectedDate`	Requires that the data provider of the list control has a field of type Date.
Image	`source`	Set an explicit row height by using the `rowHeight` property, or set the `variableRowHeight` property to `true` to size the row correctly.
Label	`text`	
NumericStepper	`value`	
Text	`text`	
TextArea	`text`	
TextInput	`text`	

In the following example, you use the NumericStepper, DateField, and CheckBox controls as the drop-in item renderers and item editors for a DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\CBInlineCellEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", contact:"John Doe",
                    quant:3, solddate:new Date(2005, 0, 1), Sent:true},
                {label1:"Order #2315", contact:"Jane Doe",
                    quant:3, solddate:new Date(2005, 0, 5), Sent:false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myDG"
            dataProvider="{myDP}"
            variableRowHeight="true"
            width="500" height="250"
            editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="label1"
                headerText="Order #"
                editable="false"/>
            <mx:DataGridColumn dataField="quant"
                headerText="Quantity"
                itemEditor="mx.controls.NumericStepper"
                editorDataField="value"/>
            <mx:DataGridColumn dataField="solddate"
                headerText="Date"
                itemRenderer="mx.controls.DateField"
                rendererIsEditor="true"
                editorDataField="selectedDate"/>
            <mx:DataGridColumn dataField="Sent"
                itemRenderer="mx.controls.CheckBox"
                rendererIsEditor="true"
                editorDataField="selected"/>
        </mx:columns >
    </mx:DataGrid>
</mx:Application>
```

To determine the field of the data provider used to populate the drop-in item renderer, Flex uses the value of the `dataField` property for the `<mx:DataGridColumn>` tag. In this example, you do the following:

- You set the `dataField` property of the second column to `quant`, and define the NumericStepper control as the item editor. Therefore, the column displays the cell value as text, and opens the NumericStepper control when you select the cell to edit it. The NumericStepper control displays the current value of the cell, and any changes you make to it are copied back to the data provider of the DataGrid control.
- You set the `dataField` property of the third column to `solddate`, and define the DateField control as the item renderer and item editor by setting the `rendererIsEditor` property to `true`. The data provider defines the data as an object of type Date, which is required to use the DateField control as the item renderer or item editor. Therefore, the column displays the date value using the DateField control, and also uses the DateField control to edit the cell value.
- You set the `dataField` property of the fourth column to `Sent`, and define the CheckBox control as the item renderer and item editor. Typically, you use `itemEditor` property to specify a different class as the item editor, and specify to use the same control as both the item renderer and item editor by setting the `rendererIsEditor` property to `true`. The CheckBox control displays a check mark in the cell if the Sent field for the row is set to `true`, and an empty check box if the field is set to `false`.

  For more information on using an item renderer as an item editor, see "Creating an editable cell" on page 904.

When you use the Image control as a drop-in item renderer, you usually have to set the row height to accommodate the image, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dropin\DGDropInImageRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                { Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: 'slanted.jpg' },
                { Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: 'brighten.jpg'}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid" dataProvider="{initDG}" rowHeight="50">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover"
                itemRenderer="mx.controls.Image"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, you use the Image control to display the album cover in a cell of the DataGrid control.

Requirements of a drop-in item renderers in a List control

When you use the `itemRenderer` property of the List control to specify a drop-in item renderer, the data in the data provider must be of the type expected by the item renderer control. For example, if you use an Image control as a drop-in item renderer, the data provider for the List control must have String data in the field.

Creating inline item renderers and editors

You define inline item renderers and item editors in the MXML definition of a component. Inline item renderers and item editors give you complete control over item rendering and cell editing.

Creating a simple inline item renderer or item editor

A simple inline item renderer contains a single control that supports the `data` property. Flex automatically copies the current cell data to the item renderer, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineImageRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: 'slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: 'brighten.jpg'}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid" dataProvider="{initDG}" rowHeight="50">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover">
                <mx:itemRenderer>
                    <mx:Component>
                        <mx:Image height="45"/>
                    </mx:Component>
                </mx:itemRenderer>
            </mx:DataGridColumn>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, you use an Image control to display the album cover.

A simple inline item editor contains a single control that supports the `data` property. Flex automatically copies the current cell data to the item renderer or item editor, and copies the new cell data back to the list control based on the value of the `editorDataField` property, as the following example item editor shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineNumStepper.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1" headerText="Order #"/>
            <mx:DataGridColumn dataField="quant" editorDataField="value">
                <mx:itemEditor>
                    <mx:Component>
                        <mx:NumericStepper stepSize="1" maximum="50"/>
                    </mx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, you return the new cell value by using the value property of the NumericStepper control.

Creating a complex inline item renderer or item editor

A complex item renderer or item editor defines multiple controls. For example, the section "Default item rendering and cell editing" on page 852 showed a DataGrid control that displayed information about albums by using three text fields. You could add a visual element to your DataGrid control to make it more compelling. To do that, you modify the data provider so that it contains a reference to a JPEG image of the album cover.

Rather than displaying the album name and album image in separate cells of the DataGrid control, you can use an inline item renderer to make them appear in a single cell, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.collections.ArrayCollection;

            // Variable in the Application scope.
            public var localVar:String="Application localVar";

            // Data includes URL to album cover.
            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover:'slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover:'brighten.jpg'}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid" dataProvider="{initDG}"
            variableRowHeight="true">
        <mx:columns>
                <mx:DataGridColumn dataField="Artist"/>
                <mx:DataGridColumn dataField="Album">
                    <mx:itemRenderer>
                        <mx:Component>
                            <mx:VBox>
                                <mx:Text id="albumName"
                                    width="100%" text="{data.Album}"/>
                                <mx:Image id="albumImage"
                                    height="45" source="{data.Cover}"/>
                            </mx:VBox>
                        </mx:Component>
                    </mx:itemRenderer>
                </mx:DataGridColumn>
                <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In the preceding example, you define three columns in the DataGrid control, and assign your item renderer to the second column. For an image that shows the output of this application, see "Using custom item renderers and item editors" on page 855.

Notice that the Text and Image controls in the item renderer both use the `data` property to initialize their values. This is necessary because you defined multiple controls in the item renderer, and Flex cannot automatically determine which data element in the data provider is associated with each control of the item renderer.

Even if the top-level container of an inline item renderer has a single child control, you must use the `data` property to initialize the child control, as the following example shows:

```
<mx:DataGridColumn dataField="Cover">
  <mx:itemRenderer>
    <mx:Component>
      <mx:VBox horizontalAlign="center">
        <mx:Image height="45" source="{data.Cover}"/>
      </mx:VBox>
    </mx:Component>
  </mx:itemRenderer>
</mx:DataGridColumn>
```

In the preceding example, you make the Image control a child of a VBox container so that you can align the image in the cell. The Image control is now a child of the VBox container, so you must initialize it by using the `data` property.

You can define multiple controls in a complex inline item editor, which lets you edit multiple values of the data provider for the list control. Alternatively, you can define a complex inline item editor so it returns a value other than a String. For more information, see Chapter 22, "Working with Item Editors," on page 903.

Items allowed in an inline component

The only restriction on what you can and cannot do in an inline item renderer or editor is that you cannot create an empty `<mx:Component></mx:Component>` tag. For example, you can combine effect and style definitions in an inline item renderer or editor along with your rendering and editing logic.

You can include the following items in an inline item renderer editor:

- Binding tags
- Effect tags
- Metadata tags
- Model tags
- Scripts tags

- Service tags
- State tags
- Styles tags
- XML tags
- `id` attributes, except for the top-most component

Using the <mx:Component> tag

You use the `<mx:Component>` tag to define an inline item renderer or item editor in an MXML file. This section describes how to use that tag.

Defining the scope in an <mx:Component> tag

The `<mx:Component>` tag defines a new scope in an MXML file, where the local scope of the item renderer or item editor is defined by the MXML code block delimited by the `<mx:Component>` and `</mx:Component>` tags. To access elements outside of the local scope of the item renderer or item editor, you prefix the element name with the `outerDocument` keyword.

For example, you define one variable named localVar in the scope of the main application, and another variable with the same name in the scope of the item renderer. From within the item renderer, you access the application's localVar by prefixing it with `outerDocument` keyword, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGImageScope.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Variable in the Application scope.
            [Bindable]
            public var localVar:String="Application localVar";

            // Data includes URL to album cover.
            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                { Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover:'slanted.jpg'},
                { Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover:'brighten.jpg'}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid" dataProvider="{initDG}" width="100%"
            variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover">
                <mx:itemRenderer>
                  <mx:Component>
                    <mx:VBox>
                        <mx:Script>
                          <![CDATA[
                            // Variable in the renderer scope.
                            [Bindable]
                        public var localVar:String="Renderer localVar";
                          ]]>
                         </mx:Script>

                         <mx:Text id="albumName"
                            width="100%"
                            selectable="false"
                            text="{data.Album}"/>
                         <mx:Image id="albumImage"
```

```
                        height="45"
                        source="{data.Cover}"/>
                    <mx:TextArea
                        text="{'Renderer localVar= ' + localVar}"/>
                    <mx:TextArea
            text="{'Application localVar= ' + outerDocument.localVar}"/>
                </mx:VBox>
            </mx:Component>
        </mx:itemRenderer>
        </mx:DataGridColumn>
        <mx:DataGridColumn dataField="Price"/>
    </mx:columns>
</mx:DataGrid>
</mx:Application>
```

One use of the `outerDocument` keyword is to initialize the data provider of a control in the inline item editor. For example, you can use a web service, or other mechanism, to pass data into the application, such as the list of U.S. states. You can then initialize all ComboBox controls that are used as item editors from a single property of the application that contains the list of U.S. states.

Specifying a class name to the inline component

You can optionally specify the `className` property of the `<mx:Component>` tag to explicitly name the class generated by Flex for the inline component. By naming the class, you define a way to reference the elements in the inline component.

For an example using the `className` property, see "Example: Passing multiple values back from an item editor" on page 925.

Creating a reusable inline item renderer or item editor

Rather than defining an inline item renderer or item editor in the definition of a component, you can define a reusable inline item renderer or item editor for use in multiple locations in your application.

For example, you use the <mx:Component> tag to define an inline item editor that consists of a ComboBox control for selecting the state portion of an address. You then use that inline item editor in two different DataGrid controls, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGEditorCBReUse.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="700" >

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', City: 'Boston', State: 'MA'},
                {Company: 'Allied', Contact: 'Jane Smith',
                  Phone: '617-555-3434', City: 'SanFrancisco', State: 'CA'}
            ]);
        ]]>
    </mx:Script>

    <mx:Component id="inlineEditor">
        <mx:ComboBox>
            <mx:dataProvider>
                <mx:String>AL</mx:String>
                <mx:String>AK</mx:String>
                <mx:String>AR</mx:String>
            </mx:dataProvider>
        </mx:ComboBox>
    </mx:Component>

    <mx:DataGrid id="myGrid"
        variableRowHeight="true"
        dataProvider="{initDG}"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="City"/>
            <mx:DataGridColumn dataField="State"
                width="150"
                editorDataField="selectedItem"
                itemEditor="{inlineEditor}"/>
        </mx:columns>
    </mx:DataGrid>

    <mx:DataGrid id="myGrid2"
```

```
        variableRowHeight="true"
        dataProvider="{initDG}"
        editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="City"/>
            <mx:DataGridColumn dataField="State"
                width="150"
                editorDataField="selectedItem"
                itemEditor="{inlineEditor}"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, you specify the `id` property of the inline item editor defined by the `<mx:Component>` tag. You then use data binding to specify the editor as the value of the itemEditor property for the two DataGrid controls.

The inline item editor or item renderer defined in the `<mx:Component>` tag appears only where you use it in the DataGrid control; otherwise Flex ignores it when laying out your application.

Creating item renderers and item editor components

Defining a custom item renderer or item editor by using an MXML component gives you greater flexibility and functionality than using a drop-in item renderer or item editor. Many of the rules for defining item renderers and item editors as custom components are the same as for using inline item renderers and editors. For more information, see "Creating inline item renderers and editors" on page 875.

This section contains information about defining custom item renderers and item editors using MXML components. For more information on working with custom components, see *Creating and Extending Flex 2 Components*.

Creating an item renderer component

The section "Default item rendering and cell editing" on page 852 showed a DataGrid control that displayed information about albums by using three text fields. You could add a visual element to your DataGrid control to make it more compelling. To do that, you modify the data provider so that it contains a URL for a JPEG image of the album cover.

The default item renderer for a DataGrid control displays data as text. To get the DataGrid control to display the image of the album cover, you use the custom item renderer defined in the RendererDGImage.mxml file, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\myComponents\RendererDGCheckBox.mxml -->
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="center" >

    <mx:Image id="albumImage" height="175" source="{data.Cover}"/>
</mx:HBox>
```

The item renderer contains an Image control in an HBox container. The HBox container specifies to center the image in the container; otherwise, the image appears flush left in the cell. The Image control specifies the height of the image as 75 pixels. By default, an image has a height of 0 pixels; therefore, if you omit the height, the image does not appear.

You use data binding to associate fields of a `data` property with the controls in an item renderer or item editor. In this example, the `data` property that is passed to the item renderer contains the element of the data provider for the entire row of the DataGrid control. You then bind the Cover field of the `data` property to the Image control.

You use a custom item renderer with the DataGrid control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\MainDGImageRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: 'slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: 'brighten.jpg'}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Cover"
                itemRenderer="myComponents.RendererDGImage"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

The DataGrid control contains a column for the album cover that uses the `itemRenderer` property to specify the name of the MXML file that contains the item renderer for that column. Now, when you run this example, the DataGrid control uses your custom item renderer for the Cover column to display an image of the Album cover.

Rather than having the album name and album image in separate cells of the DataGrid control, you can use an item renderer to make them appear in a single cell, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\myComponents\RendererDGTitleImage.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="center" height="75">

    <mx:Text id="albumName"
        width="100%"
        selectable="false"
        text="{data.Album}"/>
    <mx:Image id="albumImage"
        source="{data.Cover}"/>
</mx:VBox>
```

You save this item renderer to the RendererDGTitleImage.mxml file. The DataGrid control in the main application references the item renderer, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\MainDGTitleRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99, Cover: 'slanted.jpg'},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99, Cover: 'brighten.jpg'}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist" />
            <mx:DataGridColumn dataField="Album"
                itemRenderer="myComponents.RendererDGTitleImage" />
            <mx:DataGridColumn dataField="Price"  />
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In the preceding example, you define only three columns in the DataGrid control, and assign your item renderer to the second column. For an image that shows the output of this application, see "Using custom item renderers and item editors" on page 855.

Creating a simple item editor component

A simple item editor component defines a single control that you use to edit a cell, and returns a single value back to the list control. For an example of a simple item editor component, see "Using a component as an item renderer or item editor" on page 863.

A complex item editor can contain multiple components, or can return something other than a single value to the list control. For more information, see Chapter 22, "Working with Item Editors," on page 903.

Overriding the data property

All components that you use in a custom item renderer or item editor that require access to the data passed to the renderer must implement the mx.core.IDataRenderer interface to define the `data` property. All Flex containers and many Flex components support this property.

Classes implement the mx.core.IDataRenderer interface by defining the `data` property as a setter and getter method, with the following signature:

```
override public function set data(value:Object):void
public function get data():Object
```

In the setter method, *`value`* is the `data` property passed to the item renderer.

If you define a custom component and you want to support the `data` property, your component must implement the mx.core.IDataRenderer interface. If your component is a subclass of a class that already implements the mx.core.IDataRenderer interface, you do not have to reimplement the interface.

To add programmatic logic to the controls in your item renderer or item editor that already implement the `data` property, you can override the setter or getter method for the `data` property. Typically, you override the setter method so that you can perform some operation based on the value that is passed to it.

For example, the following DataGrid control uses `true` and `false` for the values of the SalePrice field of the data provider. Although you could display these values in your DataGrid control, you can create a more compelling DataGrid control by displaying images instead, as the following item renderer shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\component\myComponents\RendererDGImageSelect.mxml -->
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="center">

    <mx:Script>
        <![CDATA[

            import mx.events.FlexEvent;

            [Embed(source="saleIcon.jpg")]
            [Bindable]
            public var sale:Class;

            [Embed(source="noSaleIcon.jpg")]
            [Bindable]
            public var noSale:Class;

            override public function set data(value:Object):void {
                if(value != null)  {
                    super.data = value;
                    if (value.SalePrice == true) onSale.source=new sale();
                    else onSale.source=new noSale();
                }
                // Dispatch the dataChange event.
                dispatchEvent(new FlexEvent(FlexEvent.DATA_CHANGE));
            }
        ]]>
    </mx:Script>

    <mx:Image id="onSale" height="20"/>
</mx:HBox>
```

In this example, you override the setter method for the HBox container. In the override, use `super.data` to set the `data` property for the base class, and then set the `source` property of the Image control based on the value of the SalePrice field. If the field is `true`, which indicates that the item is on sale, you display one icon. If the item is not on sale, you display a different icon.

The override also dispatches the `dataChange` event to indicate that the `data` property has changed. You typically dispatch this event from the setter method.

About using the creationComplete and dataChange events

Flex dispatches the `creationComplete` event once for a component after the component is created and initialized. Many custom components define an event listener for the `creationComplete` event to handle any postprocessing tasks that must be performed after the component is completely created and initialized.

However, for an item renderer or item editor, Flex might reuse an instance of the item renderer or item editor. But, a reused instance of an item renderer or item editor does not redispatch the `creationComplete` event. Instead, you can use the `dataChange` event with an item renderer or item editor. Flex dispatches the `dataChange` event every time the `data` property changes. The example in the section "Accessing the listData property" on page 867 uses the `dataChange` event to update the TextArea in an item renderer for a DataGrid control.

Creating an item renderer in ActionScript

Although you commonly create item renderers and editors in MXML, you can also create them in ActionScript, as the following example item renderer shows:

```
package myComponents {

    // myComponents/CellField.as
    import mx.controls.*;
    import mx.core.*;
    import mx.controls.dataGridClasses.DataGridListData;

    public class CellField extends TextInput
    {
        // Define the constructor and set properties.
        public function CellField() {
            super();
            height=60;
            width=80;
            setStyle("borderStyle", "none");
            editable=false;
        }

        // Override the set method for the data property.
        override public function set data(value:Object):void {
            super.data = value;

            if (value != null)
            {
                text = value[DataGridListData(listData).dataField];
                if(Number(text) > 100)
                {
                    setStyle("backgroundColor", 0xFF0000);
                }
            }

            super.invalidateDisplayList();
        }
    }
}
```

In the preceding example, you create a subclass of the TextInput control as your item renderer. The class must be public to be used as an item renderer or editor. This item renderer displays a red background if the value of the DataGrid cell is greater than 100.

You can use this item renderer in a DataGrid control, as the following example shows:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- itemRenderers\asRenderer\MainASItemRenderer.mxml -->

<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600" height="600">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Monday: 12, Tuesday: 22, Wednesday: 452, Thursday: 90},
                {Monday: 258, Tuesday: 22, Wednesday: 45, Thursday: 46},
                {Monday: 4, Tuesday: 123, Wednesday: 50, Thursday: 95},
                {Monday: 12, Tuesday: 52, Wednesday: 111, Thursday: 20},
                {Monday: 22, Tuesday: 78, Wednesday: 4, Thursday: 51}
            ]);
        ]]>
    </mx:Script>

    <mx:Text text="All cells over 100 are red" />

    <mx:DataGrid id="myDataGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Monday"
                itemRenderer="myComponents.CellField" />
            <mx:DataGridColumn dataField="Tuesday"
                itemRenderer="myComponents.CellField" />
            <mx:DataGridColumn dataField="Wednesday"
                itemRenderer="myComponents.CellField" />
            <mx:DataGridColumn dataField="Thursday"
                itemRenderer="myComponents.CellField" />
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Working with item renderers

This section contains examples using item renderers with Flex controls. For examples using item editors, see Chapter 22, "Working with Item Editors," on page 903.

Example: Using an item renderer with the TileList and HorizontalList controls

The TileList and HorizontalList controls display a tiled list of items. The TileList control displays items in vertical columns. The HorizontalList control displays items in horizontal rows. The TileList and HorizontalList controls are particularly useful in combination with a custom item renderer for displaying a list of images and other data.

The following image shows a HorizontalList control that is used to display a product catalog:

Product images courtesy of Lavish

Each item in the HorizontalList control contains an image, a descriptive text string, and a price. The following code shows the application that displays the catalog. The item renderer for the HorizontalList control is an MXML component named Thumbnail.

```
<?xml version="1.0"?>
<!-- itemRenderers\htlistMainlistThumbnailRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Model id="catalog" source="catalog.xml"/>

    <mx:HorizontalList id="myList"
        columnWidth="125"
        rowHeight="125"
        columnCount="4"
        dataProvider="{catalog.product}"
        itemRenderer="myComponents.Thumbnail"/>

    <mx:LinkButton label="Product images courtesy of Lavish"
        click="navigateToURL(new URLRequest('http://www.shoplavish.com'),
'_blank')"/>
</mx:Application>
```

The file catalog.xml defines the data provider for the HorizontalList control:

```
<?xml version="1.0"?>
<catalog>
    <product id="1">
        <name>USB Watch</name>
        <price>129.99</price>
        <image>assets/products/usbwatch.jpg</image>
        <thumbnail>assets/products/usbwatch_sm.jpg</thumbnail>
    </product>
    <product id="2">
        <name>007 Digital Camera</name>
        <price>99.99</price>
        <image>assets/products/007camera.jpg</image>
        <thumbnail>assets/products/007camera_sm.jpg</thumbnail>
    </product>
    <product id="3">
        <name>2-Way Radio Watch</name>
        <price>49.99</price>
        <image>assets/products/radiowatch.jpg</image>
        <thumbnail>assets/products/radiowatch_sm.jpg</thumbnail>
    </product>
    <product id="4">
        <name>USB Desk Fan</name>
        <price>19.99</price>
        <image>assets/products/usbfan.jpg</image>
        <thumbnail>assets/products/usbfan_sm.jpg</thumbnail>
    </product>
    <product id="5">
        <name>Caffeinated Soap</name>
        <price>19.99</price>
        <image>assets/products/soap.jpg</image>
        <thumbnail>assets/products/soap_sm.jpg</thumbnail>
    </product>
    <product id="6">
        <name>Desktop Rovers</name>
        <price>49.99</price>
        <image>assets/products/rover.jpg</image>
        <thumbnail>assets/products/rover_sm.jpg</thumbnail>
    </product>
</catalog>
```

The following example shows the Thumbnail.mxml MXML component. In this example, you define the item renderer to contain three controls: an Image control and two Label controls. These controls examine the `data` property that is passed to the item renderer to determine the content to display.

```
<?xml version="1.0" ?>
<!-- itemRenderers\htlist\myComponents\Thumbnail.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="center"
    verticalGap="0" borderStyle="none" backgroundColor="white" >

    <mx:Image id="image" width="60" height="60" source="{data.image}"/>
    <mx:Label text="{data.name}" width="120" textAlign="center"/>
    <mx:Label text="${data.price}" fontWeight="bold"/>
</mx:VBox>
```

For more information on the TileList and HorizontalList controls, see Chapter 12, "Using Data-Driven Controls," on page 439.

Example: Using an item renderer with a DataGrid control

The DataGrid control works with the DataGridColumn class to configure the grid. For a DataGrid control, you can specify two types of renderers: one for the cells of each column, and one for the header cell at the top of each column. To specify an item renderer for a column of a DataGrid control, you use the `DataGridColumn.itemRenderer` property. To specify an item renderer for a column header cell, you use the `DataGridColumn.headerRenderer` property.

For example, to highlight a column in the DataGrid control, you can use using an icon in the column header cell to indicate a sale price, as the following item renderer shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\RendererDGHeader.mxml -->
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[

            [Embed(source="saleIcon.jpg")]
            [Bindable]
            public var sale:Class;
        ]]>
    </mx:Script>

    <mx:Label text="Sale Price!"/>
    <mx:Image height="20" source="{sale}"/>

</mx:HBox>
```

This item renderer uses a Label control to insert the text “Sale Price!” and an Image control to insert an icon in the column header.

The following example shows the main application:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainDGHeaderRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
          import mx.collections.ArrayCollection;

          [Bindable]
          private var initDG:ArrayCollection = new ArrayCollection([
            {Artist:'Pavement', Album:'Slanted and Enchanted',
                Price:11.99, SalePrice: true },
            {Artist:'Pavement', Album:'Brighten the Corners',
                Price:11.99, SalePrice: false }
          ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        variableRowHeight="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
            <mx:DataGridColumn width="150" dataField="SalePrice"
                headerRenderer="myComponents.RendererDGHeader"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, the DataGrid control displays the String `true` or `false` in the column to indicate that the price is a sale price. You can also define an item renderer for that column to display a more compelling icon rather than text. For an example that uses an item renderer with a DataGrid control, see "Using custom item renderers and item editors" on page 855.

Example: Using an item renderer with a List control

When you use a custom item renderer with a List control, you specify it using the `List.itemRenderer` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\MainListStateRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="700" width="700">

    <mx:List id="myList"
        height="180" width="250"
        variableRowHeight="true"
        itemRenderer="myComponents.RendererState"
        backgroundColor="white" >
        <mx:dataProvider>
            <mx:Object label="Alaska"
                data="Juneau"
                webPage="http://www.state.ak.us/"/>
            <mx:Object label="Alabama"
                data="Montgomery"
                webPage="http://www.alabama.gov/" />
            <mx:Object label="Arkansas"
                data="Little Rock"
                webPage="http://www.state.ar.us/"/>
        </mx:dataProvider>
    </mx:List>
</mx:Application>
```

The previous example sets the `rowHeight` property to 75 pixels because the item renderer displays content that exceeds the default row height of the List control. To see an image that shows this application, see "Using custom item renderers and item editors" on page 855.

The following item renderer, RendererState.mxml, displays the parts of each List item, and creates a LinkButton control which lets you open the state's web site:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\myComponents\RendererState.mxml -->
    <mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml" >
        <mx:Script>
            <![CDATA[
                // Import Event and URLRequest classes.
                import flash.events.Event;
                import flash.net.URLRequest;

                private var u:URLRequest;

                // Event handler to open URL using
                // the navigateToURL() method.
                private function handleClick(eventObj:Event):void {
                    u = new URLRequest(data.webPage);
                    navigateToURL(u);
                }
            ]]>
        </mx:Script>

        <mx:HBox >
            <!-- Use Label controls to display state and capital names. -->
            <mx:Label id="State" text="State: {data.label}"/>
            <mx:Label id="Statecapital" text="Capital: {data.data}" />
        </mx:HBox>

        <!-- Define the Link control to open a URL. -->
        <mx:LinkButton id="webPage" label="Official {data.label} web page"
            click="handleClick(event);" color="blue"  />
    </mx:VBox>
```

Example: Using an item renderer with a Tree control

For the Tree control, you use the `itemRenderer` property to specify a single renderer for all nodes of the tree. If you define a custom item renderer, you are responsible for handling the display of the entire node, including the text and icon.

One option is to define an item renderer as a subclass of the default item renderer class, the TreeItemRenderer class. You can then modify the item renderer as required by your application without implementing the entire renderer, as the following example shows:

```
package myComponents
{
    // itemRenderers/tree/myComponents/MyTreeItemRenderer.as
    import mx.controls.treeClasses.*;
    import mx.collections.*;
```

```
    public class MyTreeItemRenderer extends TreeItemRenderer
    {

        // Define the constructor.
        public function MyTreeItemRenderer() {
            super();
        }

        // Override the set method for the data property
        // to set the font color and style of each node.
        override public function set data(value:Object):void {
            super.data = value;
            if(TreeListData(super.listData).hasChildren)
            {
                setStyle("color", 0xff0000);
                setStyle("fontWeight", 'bold');
            }
            else
            {
                setStyle("color", 0x000000);
                setStyle("fontWeight", 'normal');
            }
        }

        // Override the updateDisplayList() method
        // to set the text for each tree node.
       override protected function updateDisplayList(unscaledWidth:Number,
            unscaledHeight:Number):void {

            super.updateDisplayList(unscaledWidth, unscaledHeight);
            if(super.data)
            {
                if(TreeListData(super.listData).hasChildren)
                {
                    var tmp:XMLList =
                        new XMLList(TreeListData(super.listData).item);
                    var myStr:int = tmp[0].children().length();
                  super.label.text =  TreeListData(super.listData).label +
                        "(" + myStr + ")";
                }
            }
        }
    }
}
```

For each node that has a child node, this item renderer displays the node text in red and includes the count of the number of child nodes in parentheses. The following example uses this item renderer in an application.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- itemRenderers\tree\MainTreeItemRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initCollections();">

    <mx:Script>
        <![CDATA[

            import mx.collections.*;

            public var xmlBalanced:XMLList =
                <>
                    <node label="Containers">
                        <node label="DividedBoxClasses">
                            <node label="BoxDivider" data="BoxDivider.as"/>
                        </node>
                        <node label="GridClasses">
                            <node label="GridRow" data="GridRow.as"/>
                            <node label="GridItem" data="GridItem.as"/>
                            <node label="Other File" data="Other.as"/>
                        </node>
                    </node>
                    <node label="Data">
                        <node label="Messages">
                            <node label="DataMessage"
                                data="DataMessage.as"/>
                            <node label="SequenceMessage"
                                data="SequenceMessage.as"/>
                        </node>
                        <node label="Events">
                            <node label="ConflictEvents"
                                data="ConflictEvent.as"/>
                            <node label="CommitFaultEvent"
                                data="CommitFaultEvent.as"/>
                        </node>
                    </node>
                </>;

            [Bindable]
            public var xlcBalanced:XMLListCollection;

            private function initCollections():void {
                xlcBalanced = new XMLListCollection(xmlBalanced);
            }
        ]]>
    </mx:Script>
```

```
    <mx:Text width="400"
        text="The nodes with children are in bold red text, with the number
of children in parenthesis.)"/>

    <mx:Tree id="compBalanced"
        width="400" height="500"
        dataProvider="{xlcBalanced}"
        labelField="@label"
        itemRenderer="myComponents.MyTreeItemRenderer"/>
</mx:Application>
```

CHAPTER 22

Working with Item Editors

22

Item editors let you modify the value of a cell of a list control. The DataGrid, List, and Tree controls support item editors. This topic describes the cell editing process, the events associated with cell editing, and includes examples of ways you can create and use item editors.

For an introduction to item renderers and item editors, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

Contents

The cell editing process

The following set of steps occur when a cell in a list control is edited:

1. User releases the mouse button while over a cell, tabs to a cell, or in another way attempts to edit a cell.
2. Adobe Flex dispatches the `itemEditBeginning` event. You can use this event to disable editing of a specific cell or cells. For more information, see "Example: Preventing a cell from being edited" on page 921.

3. Flex dispatches the `itemEditBegin` event to open the item editor. You can use this event to modify the data passed to the item editor. For more information, see "Example: Modifying data passed to or received from an item editor" on page 923.
4. The user edits the cell.
5. The user ends the editing session. Typically the cell editing session ends when the user removes focus from the cell.
6. Flex dispatches the `itemEditEnd` event to close the item editor and update the list control with the new cell data. You can use this event to modify the data returned to the cell, validate the new data, or return data in a format other than the format returned by the item editor. For more information, see "Example: Modifying data passed to or received from an item editor" on page 923.
7. The new data value appears in the cell.

Creating an editable cell

The DataGrid, List, and Tree controls include an `editable` property that you set to `true` to let users edit the contents of the control. By default, the value of the `editable` property is `false`, which means that you cannot edit the cells. For a DataGrid control, setting the `editable` property to `true` enables editing for all columns of the grid. You can disable editing for any column by setting the `DatagridColumn.editable` property to `false`.

Your list controls can use the default item editor (TextInput control), a custom item editor, or a custom item renderer as an editor. The `rendererIsEditor` property of the list controls determines whether you can use an item renderer as an item editor, as the following table shows:

rendererIsEditor property	itemRenderer property	itemEditor property
`false` (default)	Specifies the item renderer.	Specifies the item editor. Selecting the cell opens the item editor as defined by the `itemEditor` property. If the `itemEditor` property is undefined, use the default item editor (TextInput control).
`true`	Specifies the item renderer to display the cell contents. You can use the item renderer as an item editor.	Ignored.

As this table shows, the state of the `rendererIsEditor` property defines whether to use a custom item renderer as the item editor, or a custom item editor. If you set the `rendererIsEditor` property to `true`, Flex uses the item renderer as an item editor, and ignores the `itemEditor` property.

For an example, see "Example: Using an item renderer as an item editor" on page 931.

Returning data from an item editor

By default, Flex expects an item editor to return a single value to the list control. You use the `editorDataField` property of the list control to specify the property of the item editor that contains the new data. Flex converts the value to the appropriate data type for the cell.

The default item editor is a TextInput control. Therefore, the default value of the `editorDataField` property is `"text"`, which corresponds to the `text` property of the TextInput control. If you specify a custom item editor, you also set the `editorDataField` property to the appropriate property of the item editor, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dropin\DropInNumStepper.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection([
                {label1:"Order #2314", quant:3, Sent:true},
                {label1:"Order #2315", quant:3, Sent:false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myDG" dataProvider="{myDP}"
        variableRowHeight="true"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="label1"
                headerText="Order #" />
            <mx:DataGridColumn dataField="quant"
                itemEditor="mx.controls.NumericStepper"
                editorDataField="value" />
        </mx:columns >
    </mx:DataGrid>
</mx:Application>
```

In the preceding example, you use a NumericStepper control as the item editor, and therefore, set the `editorDataField` property to `"value"`, the property of the NumericStepper that contains the new cell data.

For information about returning multiple values from an item editor, see "Example: Passing multiple values back from an item editor" on page 925.

Defining a property to return data

An item editor can contain more than a single component. For example, the following item editor contains a parent VBox container with a child CheckBox control used to edit the cell:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\EditorDGCheckBox.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
     backgroundColor="yellow">

    <mx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            public var cbSelected:Boolean;
        ]]>
    </mx:Script>

    <mx:CheckBox id="followUpCB" label="Follow up needed"
        height="100%" width="100%"
        selected="{data.FollowUp}"
        click="cbSelected=followUpCB.selected"
        updateComplete="cbSelected=followUpCB.selected;"/>
</mx:VBox>
```

In the preceding example, when the user selects the cell, Flex displays the CheckBox control in a yellow background, as defined by the parent VBox container. The user then selects or deselects the CheckBox control to set the new value for the cell.

To return a value to the list control, the VBox container defines a new property named `cbSelected`. This is necessary because you can only set the `editorDataField` property to a property of the top-level component of the item editor. That means you cannot set `editorDataField` to the `selected` property of the CheckBox control when it is a child of the VBox container.

You use the `updateComplete` event to set the value of the `cbSelected` property in case the user selects the cell to open the CheckBox control, but does not change the state of the CheckBox control.

In the following example, you use this item editor in a DataGrid control that displays a list of customer contacts. The list includes the company name, contact name, and phone number, and a column that indicates whether you must follow up with that contact:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainDGCheckBoxEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                   Phone: '413-555-1212', Date: '5/5/05' , FollowUp: true },
                {Company: 'Allied', Contact: 'Jane Smith',
                   Phone: '617-555-3434', Date: '5/6/05' , FollowUp: false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="Date"/>
            <mx:DataGridColumn dataField="FollowUp"
                width="150"
                headerText="Follow Up?"
                itemEditor="myComponents.EditorDGCheckBox"
                editorDataField="cbSelected"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Notice that the `editorDataField` property is set to `cbSelected`, the new property of the VBox container.

You also use this same mechanism with an inline item renderer that contains multiple controls. For example, you can modify the previous DataGrid example to use an inline item editor, rather than an item editor component, as the following code shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGCheckBoxEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                  Phone: '413-555-1212', Date: '5/5/05' , FollowUp: true },
                {Company: 'Allied', Contact: 'Jane Smith',
                  Phone: '617-555-3434', Date: '5/6/05' , FollowUp: false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="Date"/>
            <mx:DataGridColumn dataField="FollowUp"
                width="150"
                headerText="Follow Up?"
                editorDataField="cbSelected">

                <mx:itemEditor>
                    <mx:Component>
                        <mx:VBox backgroundColor="yellow">
                            <mx:Script>
                                <![CDATA[
                                    // Define a property for returning
                                    // the new value to the cell.
                                    [Bindable]
                                    public var cbSelected:Boolean;
                                ]]>
                            </mx:Script>

                            <mx:CheckBox id="followUpCB"
                                label="Follow up needed"
                                height="100%" width="100%"
```

```
                        selected="{data.FollowUp}"
                        click="cbSelected=followUpCB.selected"/>
                </mx:VBox>
            </mx:Component>
        </mx:itemEditor>
    </mx:DataGridColumn>
</mx:columns>
</mx:DataGrid>
</mx:Application>
```

Sizing and positioning an item editor

When an item editor appears, it appears as a pop-up control above the selected cell of the list control. The list control also hides the current value of the cell.

The CheckBox control in the previous example, "Defining a property to return data" on page 906, sizes itself to 100% for both `width` and `height`, and the VBox container sets its `backgroundColor` style property to yellow. By sizing the item editor to the size of the cell, and by setting the background color of the container, you completely cover the underlying cell.

The following table describes the properties of the list controls that you can use to size the item editor:

Property	Description
`editorHeightOffset`	Specifies the height of the item editor, in pixels, relative to the size of the cell for a DataGridColumn control, or the text field of a Tree control.
`editorWidthOffset`	Specifies the width of the item editor, in pixels, relative to the size of the cell for a DataGridColum control, or the text field of a Tree control.
`editorXOffset`	Specifies the *x* location of the upper-left corner of the item editor, in pixels, relative to the upper-left corner of the cell for a DataGridColumn control, or from the upper-left corner of the text field of a Tree control.
`editorYOffset`	Specifies the *y* location of the upper-left corner of the item editor, in pixels, relative to the upper-left corner of the cell for a DataGridColumn control, or from the upper-left corner of the text field of a Tree control.

The following code modifies the definition of the DataGridColumn control from the previous section to use the `editorXOffset` and `editorYOffset` properties to move the item editor down and to the right by 15 pixels so that it has a more prominent appearance in the DataGrid control:

```
<?xml version="1.0"?>
<!-- itemRenderers\inline\InlineDGCheckBoxEditorWithOffsets.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                   Phone: '413-555-1212', Date: '5/5/05' , FollowUp: true },
                {Company: 'Allied', Contact: 'Jane Smith',
                   Phone: '617-555-3434', Date: '5/6/05' , FollowUp: false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true" >
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="Date"/>
            <mx:DataGridColumn dataField="FollowUp"
                width="150"
                headerText="Follow Up?"
                editorDataField="cbSelected"
                editorXOffset="15"
                editorYOffset="15">

                <mx:itemEditor>
                    <mx:Component>
                        <mx:VBox backgroundColor="yellow">
                            <mx:Script>
                                <![CDATA[
                                    // Define a property for returning
                                    // the new value to the cell.
                                    [Bindable]
                                    public var cbSelected:Boolean;
                                ]]>
                            </mx:Script>
```

```
                            <mx:CheckBox id="followUpCB"
                                label="Follow up needed"
                                height="100%" width="100%"
                                selected="{data.FollowUp}"
                                click="cbSelected=followUpCB.selected"/>
                        </mx:VBox>
                    </mx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Making an item editor that responds to the Enter key

When you use the default item editor in a list control, you can edit the cell value, and then press the Enter key to move focus to the next cell in the control. When you create a simple item editor that contains only a single component, and that component implements the IFocusable interface, the item editor also responds to the Enter key. The following components implement the IFocusable interface: Accordion, Button, ButtonBar, ComboBase, DateChooser, DateField, ListBase, MenuBar, NumericStepper, TabNavigator, TextArea, and TextInput.

In the following example, you define a complex item renderer with a VBox container as the top-level component and a CheckBox control as its child component:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\EditorDGCheckBox.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
     backgroundColor="yellow">

    <mx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            public var cbSelected:Boolean;
        ]]>
    </mx:Script>

    <mx:CheckBox id="followUpCB" label="Follow up needed"
        height="100%" width="100%"
        selected="{data.FollowUp}"
        click="cbSelected=followUpCB.selected"
        updateComplete="cbSelected=followUpCB.selected;"/>
</mx:VBox>
```

When this item editor opens, the CheckBox control can obtain focus for editing, but pressing the Enter key does not move focus to the next cell because the parent VBox container does not implement the IFocusManagerComponent interface. You can modify this example to implement the IFocusManagerComponent interface, as the following code shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\EditorDGCheckBoxFocusable.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
     backgroundColor="yellow"
     implements="mx.managers.IFocusManagerComponent" >

    <mx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            public var cbSelected:Boolean;

            // Implement the drawFocus() method for the VBox.
            override public function drawFocus(isFocused:Boolean):void {
                // This method can be empty, or you can use it
                // to make a visual change to the component.
            }
        ]]>
    </mx:Script>

    <mx:CheckBox id="followUpCB"
        label="Follow up needed"
        height="100%" width="100%"
        selected="{data.FollowUp}"
        click="cbSelected=followUpCB.selected"
        updateComplete="cbSelected=followUpCB.selected;"/>
</mx:VBox>
```

While the IFocusManagerComponent interface defines several properties and methods, the UIComponent class defines or inherits implementations for all of them except for the `drawFocus()` method. Therefore, you only have to implement that one method so that your item editor responds to the Enter key.

Using the cell editing events

A list component dispatches three events as part of the cell editing process: the `itemEditBeginning`, `itemEditBegin`, and the `itemEditEnd` events. The list controls define default event listeners for all three of these events.

You can write your own event listeners for one or more of these events to customize the editing process. When you write your own event listener, it executes before the default event listener, which is defined by the component, and then the default listener executes. For example, you can modify the data passed to or returned from an item editor. You can modify the data in the event listener for the `itemEditBegin` event. When it completes, the default event listener runs to continue the editing process.

However, you can replace the default event listener for the component with your own event listener. To prevent the default event listener from executing, you call the `preventDefault()` method from anywhere in your event listener.

You use the following events when you create an item editor:

- `itemEditBeginning`

  Dispatched when the user releases the mouse button while over a cell, tabs to a cell, or in any way attempts to edit a cell.

  The list controls have a default listener for the `itemEditBeginning` event that sets the `editedItemPosition` property of the list control to the cell that has focus.

  You typically write your own event listener for this event to prevent editing of a specific cell or cells. To prevent editing, call the `preventDefault()` method from within your event listener, which stops the default event listener from executing, and prevents any editing from occurring on the cell. For more information, see "Example: Preventing a cell from being edited" on page 921.

- `itemEditBegin`

  Dispatched before an item editor opens.

  The list components have a default listener for the `itemEditBegin` event that calls the `createItemEditor()` method to perform the following actions:

  - Creates an item editor object, and copies the `data` property from the cell to the editor. By default, the item editor object is an instance of the TextInput control. You use the `itemEditor` property of the list control to specify a custom item editor class.
  - Sets the `itemEditorInstance` property of the list control to reference the item editor object.

  You can write an event listener for this event to modify the data passed to the item editor. For example, you might modify the data, its format, or other information used by the item editor. For more information, see "Example: Modifying data passed to or received from an item editor" on page 923.

You can also create an event listener to determine which item editor you use to edit the cell. For example, you might have two different item editors. Within the event listener, you can examine the data to be edited, open the appropriate item editor by setting the `itemEditor` property to the appropriate editor, and then call the `createItemEditor()` method. In this case, first you call `preventDefault()` to stop Flex from calling the `createItemEditor()` method as part of the default event listener.

You can only call the `createItemEditor()` method from within the event listener for the `itemEditBegin` event. To create an editor at other times, set the `editedItemPosition` property to generate the `itemEditBegin` event.

- `itemEditEnd`

  Dispatched when the cell editing session ends, typically when focus is removed from the cell.

  The list components have a default listener for this event that copies the data from the item editor to the data provider of the list control. The default event listener performs the following actions:

  - Uses the `editorDataField` property of the list control to determine the property of the item editor that contains the new data. The default item editor is the TextInput control, so the default value of the `editorDataField` property is `"text"` to specify that the `text` property of the TextInput control contains the new cell data.
  - Depending on the reason for ending the editing session, the default event listener calls the `destroyItemEditor()` method to close the item editor. For more information, see "Determining the reason for an itemEditEnd event" on page 917.

  You typically write an event listener for this event to perform the following actions:

  - Modify the data returned from the item editor.

    In your event listener, you can modify the data returned by the editor to the list control. For example, you can reformat the data before returning it to the list control. By default, an item editor can return only a single value. You must write an event listener for the `itemEditEnd` event if you want to return multiple values.

  - Examine the data entered into the item editor

    In your event listener, you can examine the data entered into the item editor. If the data is incorrect, you can call the `preventDefault()` method to stop Flex from passing the new data back to the list control and from closing the editor.

Cell editing event classes

Each editable list control has a corresponding class that defines the event object for the cell editing events, as the following table shows:

List class	Event class
DataGrid	DataGridEvent
List	ListEvent
Tree	ListEvent

Notice that the event class for the List and Tree controls is ListEvent.

When defining the event listener for a list control, ensure that you specify the correct type for the event object passed to the event listener, as the following example shows for a DataGrid control:

```
public function myCellEndEvent(event:DataGridEvent):void {
  // Define event listener.
}
```

Accessing cell data and the item editor in an event listener

From within an event listener, you can access the current value of the cell being edited, the new value entered by the user, or the item editor used to edit the cell.

To access the current value of a cell, you use the `editedItemRenderer` property of the list control. The `editedItemRenderer` property contains the data that corresponds to the cell being edited. For List and Tree controls, this property contains the data provider element for the cell. For a DataGrid control, it contains the data provider element for the entire row of the DataGrid.

To access the new cell value and the item editor, you use the `itemEditorInstance` property of the list control. The `itemEditorInstance` property is not initialized until after the event listener for the `cellBeginEvent` listener executes. Therefore, you typically only access the `itemEditorInstance` property from within the event listener for the `itemEditEnd` event.

The following example shows an event listener for the `itemEditEnd` event that uses these these properties:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventAccessEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.controls.TextInput;
            import mx.events.DataGridEvent;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99 }
            ]);

            // Define event listener for the itemEditEnd event.
            private function getCellInfo(event:DataGridEvent):void {

                // Get the cell editor and cast it to TextInput.
                var myEditor:TextInput =
                    TextInput(event.currentTarget.itemEditorInstance);

                // Get the new value from the editor.
                var newVal:String = myEditor.text;

                // Get the old value.
                var oldVal:String =
            event.currentTarget.editedItemRenderer.data[event.dataField];

                // Write out the cell coordinates, new value,
                // and old value to the TextArea control.
                cellInfo.text = "cell edited.\n";
                cellInfo.text += "Row, column: " + event.rowIndex + ", " +
                    event.columnIndex + "\n";
                cellInfo.text += "New value: " + newVal + "\n";
                cellInfo.text += "Old value: " + oldVal;
            }
        ]]>
    </mx:Script>

    <mx:TextArea id="cellInfo" width="300" height="150" />

    <mx:DataGrid id="myGrid"
```

```
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="getCellInfo(event);" >
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, you access the item editor, and cast it to the correct editor class. The default item editor is a TextInput control, so you cast it to TextInput. If you had defined a custom item editor, you would cast it to that class. After you have a reference to the item editor, you can access its properties to obtain the new cell value.

To access the old value of the cell, you use the `editedItemRenderer` property of the DataGrid control. You then use the `dataField` property of the event object to access the `data` property for the edited column.

Determining the reason for an itemEditEnd event

A user can end a cell editing session in several ways. In the body of the event listener for the `itemEditEnd` event, you can determine the reason for the event, and then handle it accordingly.

Each event class for a list control defines the `reason` property, which contains a value that indicates the reason for the event. The `reason` property has the following values:

Value	Description
`CANCELLED`	Specifies that the user canceled editing and that they do not want to save the edited data. Even if you call the `preventDefault()` method from within your event listener for the `itemEditEnd` event, Flex still calls the `destroyItemEditor()` method to close the editor.
`NEW_COLUMN`	(DataGrid only) Specifies that the user moved focus to a new column in the same row. In an event listener, you can let the focus change occur, or prevent it. For example, your event listener might check that the user entered a valid value for the cell currently being edited. If not, you can prevent the user from moving to a new cell by calling the `preventDefault()` method. In this case, the item editor remains open, and the user continues to edit the current cell. If you call the `preventDefault()` method and also call the `destroyItemEditor()` method, you block the move to the new cell, but the item editor closes.

Value	Description
NEW_ROW	Specifies that the user moved focus to a new row. You handle this value for the reason property similar to the way you handle the NEW_COLUMN value.
OTHER	Specifies that the list control lost focus, was scrolled, or is somehow in a state where editing is not allowed. Even if you call the preventDefault() method from within your event listener for the itemEditEnd event, Flex still calls the destroyItemEditor() method to close the editor.

The following example uses the `itemEditEnd` event to ensure that the user did not enter an empty String in a cell. If there is an empty String, the `itemEditEnd` event calls `preventDefault()` method to prohibit the user from removing focus from the cell until the user enters a valid value. However, if the `reason` property for the `itemEditEnd` event has the value `CANCELLED`, the event listener does nothing:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.controls.TextInput;
            import mx.events.DataGridEvent;
            import mx.events.DataGridEventReason;
            import mx.formatters.NumberFormatter;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99 }
            ]);

            private var myFormatter:NumberFormatter=new NumberFormatter();

         public function formatData(event:DataGridEvent):void {
                // Check the reason for the event.
                if (event.reason == DataGridEventReason.CANCELLED)
                {
                    // Do not update cell.
                    return;
                }

                // Get the new data value from the editor.
                var newData:String=
                    TextInput(event.currentTarget.itemEditorInstance).text;

                if(newData == "")
                {
                    // Prevent the user from removing focus,
                    // and leave the cell editor open.
                    event.preventDefault();
                    // Write a message to the errorString property.
                    // This message appears when the user
                    // mouses over the editor.
```

```
                    TextInput(myGrid.itemEditorInstance).errorString=
                        "Enter a valid string.";
                }
            }

        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="formatData(event);">
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In this example, if the user's reason for the event is `CANCELLED`, the event listener does nothing.

If the reason is `NEW_COLUMN`, `NEW_ROW`, or `OTHER`, the event listener performs the following actions:

1. Checks if the new cell value is an empty String.
2. If it is an empty String, the event listener calls the `preventDefault()` method to prevent Flex from closing the editor and from updating the cell with the empty String.
3. Writes a message to the `errorString` property of the TextInput control. This message causes a red box to appear around the TextInput control, and the message appears as a tooltip when the user moves the mouse over the cell.

Item editor examples

This section contains the following item editor examples:

- "Example: Preventing a cell from being edited" on page 921
- "Example: Modifying data passed to or received from an item editor" on page 923
- "Example: Passing multiple values back from an item editor" on page 925
- "Example: Using an item renderer as an item editor" on page 931
- "Example: Using a data validator in a custom item editor" on page 933

Example: Preventing a cell from being edited

From within an event listener for the `itemEditBeginning` event, you can inspect the cell being edited, and prevent the edit from occurring. This technique is useful when you want to prevent editing of a specific cell or cells, but allow editing of other cells.

For example, the DataGrid control uses the `editable` property to make all cells in the DataGrid control editable. You can override that for a specific column, using the `editable` property of a DataGridColumn, but you cannot enable or disable editing for specific cells.

To prevent cell editing, call the `preventDefault()` method from within your event listener for the `itemEditBeginning` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventPreventEdit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.events.DataGridEvent;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99}
            ]);

            // Define event listener for the cellEdit event
            // to prohibit editing of the Album column.
            private function disableEditing(event:DataGridEvent):void {
                if(event.columnIndex==1)
                {
                    event.preventDefault();
                }
            }

        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true"
        itemEditBeginning="disableEditing(event);" >
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Although the preceding example uses the column index, you could inspect any property of the DataGrid, or of the cell being edited, to make your decision about allowing the user to edit the cell.

Example: Modifying data passed to or received from an item editor

You can use the `itemEditBegin` and `itemEditEnd` events to examine the data passed to and from the item editor, and modify it if necessary. For example, you could reformat the data, extract a part of the data for editing, or examine the data to validate it.

You could use one of the Flex formatter classes to format data returned from an item editor. In the following example, you let the user edit the Price column of a DataGrid control. You then define an event listener for the `itemEditEnd` event that uses the NumberFormatter class to format the new cell value so that it contains only two digits after the decimal point, as the following code shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\EndEditEventFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[

            import mx.controls.TextInput;
            import mx.events.DataGridEvent;
            import mx.events.DataGridEventReason;
            import mx.formatters.NumberFormatter;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {Artist:'Pavement', Album:'Slanted and Enchanted',
                    Price:11.99},
                {Artist:'Pavement', Album:'Brighten the Corners',
                    Price:11.99 }
            ]);

            // Define the number formatter.
            private var myFormatter:NumberFormatter=new NumberFormatter();

            // Define the eventlistner for the itemEditEnd event.
            public function formatData(event:DataGridEvent):void {
                // Check the reason for the event.
                if (event.reason == DataGridEventReason.CANCELLED)
                {
                    // Do not update cell.
                    return;
                }

                // Get the new data value from the editor.
                var newData:String=
                    TextInput(event.currentTarget.itemEditorInstance).text;

                // Determine if the new value is an empty String.
                if(newData == "") {
                    // Prevent the user from removing focus,
                    // and leave the cell editor open.
                    event.preventDefault();
                    // Write a message to the errorString property.
```

```
                    // This message appears when the user
                    // mouses over the editor.
                    TextInput(myGrid.itemEditorInstance).errorString=
                        "Enter a valid string.";
                    return;
                }

                // For the Price column, return a value
                // with a precision of 2.
                if(event.dataField == "Price") {
                    myFormatter.precision=2;
                    TextInput(myGrid.itemEditorInstance).text=
                        myFormatter.format(newData);
                }
            }
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="formatData(event);" >
        <mx:columns>
            <mx:DataGridColumn dataField="Artist"/>
            <mx:DataGridColumn dataField="Album"/>
            <mx:DataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Example: Passing multiple values back from an item editor

The cell editing mechanism is optimized to work with item editors that return a single value to the list control. In the definition of the list control, you use the `editorDataField` property to specify the property of the item editor that contains the new cell data.

However, you might create an item editor that returns data in a format other than a single value. You could return multiple values, either multiple scalar values or an object containing the values.

To return data other than a single value, you write an event listener for the `cellEndEvent` that takes data from the item editor and writes it directly to the `editedItemRenderer` property of the list control. Writing it to the `editedItemRenderer` property also updates the corresponding data provider for the list control with the new data.

The following example shows an item editor implemented as a component that uses a TextInput control and a ComboBox control to let the user set the city and state portions of an address:

```
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="yellow">
<!-- itemRenderers\events\myComponents\CityStateEditor.mxml -->

    <mx:TextInput id="setCity"  width="130" text="{data.City}"/>

    <mx:ComboBox id="pickState" selectedItem="{data.State}">
        <mx:dataProvider>
            <mx:String>AL</mx:String>
            <mx:String>AK</mx:String>
            <mx:String>AR</mx:String>
            <mx:String>CA</mx:String>
            <mx:String>MA</mx:String>
        </mx:dataProvider>
    </mx:ComboBox>
</mx:VBox>
```

In the event listener for the `cellEndEvent` event, you access the new data in the TextInput and ComboBox controls by using the `itemEditorInstance` property of the list control. Then, you can write those new values directly to the `editedItemRenderer` property to update the list control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\ComplexDGEditorReturnObject.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="700">

    <mx:Script>
        <![CDATA[

            import mx.events.DataGridEvent;
            import mx.events.DataGridEventReason;
            import mx.collections.ArrayCollection;
            import myComponents.CityStateEditor;

            [Bindable]
            public var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', City: 'Boston', State: 'MA'},
                {Company: 'Allied', Contact: 'Jane Smith',
                  Phone: '617-555-3434', City: 'SanFrancisco', State: 'CA'}
             ]);

            // Define the event listener.
            public function processData(event:DataGridEvent):void {
                // Check the reason for the event.
                if (event.reason == DataGridEventReason.CANCELLED){
                    // Do not update cell.
                    return;
                }

                if(event.dataField == "City and State")
                {
                    // Disable copying data back to the control.
                    event.preventDefault();

                    // Get new city from editor.
                    myGrid.editedItemRenderer.data.City =
CityStateEditor(DataGrid(event.target).itemEditorInstance).setCity.text;

                    // Get new state from editor.
                    myGrid.editedItemRenderer.data.State =
CityStateEditor(DataGrid(event.target).itemEditorInstance).pickState.select
edItem;

                    // Close the cell editor.
                    myGrid.destroyItemEditor();
```

```
                    // Notify the list control to update its display.
                    myGrid.dataProvider.itemUpdated(event.itemRenderer.data);
                }
            }
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        rowHeight="75"
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="processData(event);">
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="City and State" width="150"
                    itemEditor="myComponents.CityStateEditor">
                <mx:itemRenderer>
                    <mx:Component>
                        <mx:Text selectable="false" width="100%"
                            text="{data.City}, {data.State}"/>
                    </mx:Component>
                </mx:itemRenderer>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Notice that the preceding example uses an inline item renderer to display the city and state in a single column of the DataGrid control.

The event listener for the `itemEditEnd` event determines if the column being edited is the city and state column. If so, the event listener performs the following actions:

1. Calls the `preventDefault()` method to prevent Flex from returning a String value.
2. Obtains the new city and state values from the item editor using the `itemEditorInstance` property of the list control.
3. Calls the `destroyItemEditor()` method to close the item editor.
4. Calls the `itemUpdated()` method to cause the list control to update its display based on the new data passed to it. If you do not call this method, the new data does not appear until the next time the list control updates its appearance.

For examples of using the List and Tree controls, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

The following example performs the same action, but uses an inline item editor, rather than a component:

```
<?xml version="1.0"?>
<!-- itemRenderers\events\InlineDGEditorReturnObject.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="700">

    <mx:Script>
        <![CDATA[

            import mx.events.DataGridEvent;
            import mx.collections.ArrayCollection;
            import mx.controls.TextInput;

            public var newCity:String;
            public var newState:String;

            [Bindable]
            public var initDG:ArrayCollection = new ArrayCollection([
                {Company: 'Acme', Contact: 'Bob Jones',
                    Phone: '413-555-1212', City: 'Boston', State: 'MA'},
                {Company: 'Allied', Contact: 'Jane Smith',
                  Phone: '617-555-3434', City: 'SanFrancisco', State: 'CA'}
             ]);

         public function processData(event:DataGridEvent):void {
                if(event.dataField=='City/State')
                {
                    // Disable copying data back to the control.
                    event.preventDefault();

                    myGrid.editedItemRenderer.data.City=
                        myEditor(myGrid.itemEditorInstance).setCity.text;
                    myGrid.editedItemRenderer.data.State=

myEditor(myGrid.itemEditorInstance).pickState.selectedItem;

                    myGrid.destroyItemEditor();

                    // Notify the list control to update its display.

myGrid.dataProvider.itemUpdated(myGrid.editedItemRenderer);
                }
            }
        ]]>
    </mx:Script>

    <mx:DataGrid id="myGrid"
        rowHeight="75"
```

```
        dataProvider="{initDG}"
        editable="true"
        itemEditEnd="processData(event);">
        <mx:columns>
            <mx:DataGridColumn dataField="Company" editable="false"/>
            <mx:DataGridColumn dataField="Contact"/>
            <mx:DataGridColumn dataField="Phone"/>
            <mx:DataGridColumn dataField="City/State" width="150">
                <mx:itemRenderer>
                    <mx:Component>
                        <mx:Text selectable="false" width="100%"
                            text="{data.City}, {data.State}"/>
                    </mx:Component>
                </mx:itemRenderer>

                <mx:itemEditor>
                    <mx:Component className="myEditor">
                        <mx:VBox backgroundColor="yellow">

                            <mx:TextInput id="setCity" width="130"
                                text="{data.City}"/>

                            <mx:ComboBox id="pickState"
                                selectedItem="{data.State}">
                                    <mx:dataProvider>
                                        <mx:String>AL</mx:String>
                                        <mx:String>AK</mx:String>
                                        <mx:String>AR</mx:String>
                                        <mx:String>CA</mx:String>
                                        <mx:String>MA</mx:String>
                                    </mx:dataProvider>
                            </mx:ComboBox>
                        </mx:VBox>
                    </mx:Component>
                </mx:itemEditor>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

The `<mx:Component>` tag includes a `className` property with a value of `"myEditor"`. The `className` property defines the name of the class that Flex creates to represent the inline item editor, just as the name of the MXML file in the previous example defines the name of the class for the component item editor. In the event listener for the `itemEditEnd` event, you can access the TextInput and ComboBox controls as properties of the myEditor class.

Example: Using an item renderer as an item editor

If you set the `rendererIsEditor` property of the DataGrid, List, or Tree control to `true`, the control uses the default TextInput control as the item editor, or the item renderer that specifies the `itemRenderer` property. If you specify an item renderer, you must ensure that you include editable controls in it so that the user can edit values.

For example, the following item renderer displays information in the cell by using the TextInput control, and lets the user edit the cell's contents:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\myComponents\MyContactEditable.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml" >

   <mx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            [Bindable]
            public var newContact:String;
        ]]>
    </mx:Script>

    <mx:Label id="title" text="{data.label1}"/>
    <mx:Label id="contactLabel" text="Last Contacted By:"/>
    <mx:TextInput id="contactTI"
        editable="true"
        text="{data.Contact}"
        change="newContact=contactTI.text;"/>
</mx:VBox>
```

You can use this item renderer with a DataGrid control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\dataGrid\MainAppEditable.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="700" width="700">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {label1: "Order #2314", Contact: "John Doe",
                    Confirmed: false, Photo: "john_doe.jpg", Sent: false},
                {label1: "Order #2315", Contact: "Jane Doe",
                    Confirmed: true, Photo: "jane_doe.jpg", Sent: false}
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="myDG"
        width="500" height="250"
        dataProvider="{initDG}"
        variableRowHeight="true"
        editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="Photo"
                editable="false"/>
            <mx:DataGridColumn dataField="Contact"
                width="200"
                editable="true"
                rendererIsEditor="true"
                itemRenderer="myComponents.MyContactEditable"
                editorDataField="newContact"/>
            <mx:DataGridColumn dataField="Confirmed"
                editable="true"
                rendererIsEditor="true"
                itemRenderer="mx.controls.CheckBox"
                editorDataField="selected"/>
            <mx:DataGridColumn dataField="Sent"
                editable="true"
                rendererIsEditor="false"
                itemEditor="mx.controls.CheckBox"
                editorDataField="selected"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

In the previous example, you use the item renderer as the item editor by setting the `rendererIsEditor` property to `true` in the second and third columns of the DataGrid control.

Example: Using a data validator in a custom item editor

Just as you can validate data in other types of controls, you can validate data in the cells of list-based controls. To do so, you can create an item renderer or item editor that incorporates a data validator. For more information about data validators, see Chapter 40, "Validating Data," on page 1281.

The following example shows the code for the validating item editor component. It uses a TextInput control to edit the field. In this example, you assign a PhoneNumberValidator validator to the `text` property of the TextInput control to validate the user input:

```
<?xml version="1.0"?>
<!-- itemRenderers\validator\myComponents\EditorPhoneValidator.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Script>
        <![CDATA[
            // Define a property for returning the new value to the cell.
            [Bindable]
            public var returnPN:String;
        ]]>
    </mx:Script>

    <mx:PhoneNumberValidator id="pnV"
        source="{newPN}"
        property="text"
        trigger="{newPN}"
        triggerEvent="change"
        required="true"/>
    <mx:TextInput id="newPN"
        text="{data.phone}"
        updateComplete="returnPN=newPN.text;"
        change="returnPN=newPN.text;"/>
</mx:VBox>
```

If the user enters an incorrect phone number, the PhoneNumberValidator draws a red box around the editor and shows a validation error message when the user moves the mouse over it.

The following example shows the code for the application that uses this item editor:

```
<?xml version="1.0" ?>
<!-- itemRenderers\validator\MainDGValidatorEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            private var initDG:ArrayCollection = new ArrayCollection([
                {name: 'Bob Jones', phone: '413-555-1212',
                    email: 'bjones@acme.com'},
                {name: 'Sally Smith', phone: '617-555-5833',
                    email: 'ssmith@acme.com'},
            ]);
        ]]>
    </mx:Script>

    <mx:DataGrid id="dg"
        width="500" height="200"
        editable="true"
        dataProvider="{initDG}">
        <mx:columns>
            <mx:DataGridColumn dataField="name"
                headerText="Name" />
            <mx:DataGridColumn dataField="phone"
                headerText="Phone"
                itemEditor="myComponents.EditorPhoneValidator"
                editorDataField="returnPN"/>
            <mx:DataGridColumn dataField="email" headerText="Email" />
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Examples using item editors with the list controls

This section describes how to create item editors for the DataGrid, List, and Tree controls. For more information on item editors, see Chapter 22, "Working with Item Editors," on page 903.

Example: Using an item editor with a DataGrid control

For examples of using item editors with the DataGrid control, see Chapter 22, "Working with Item Editors," on page 903.

Example: Using a custom item editor with a List control

You can add an item editor to a List control to let users edit the information for each state, as the following item editor shows:

```
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml">
<!-- itemRenderers\list\myComponents\EditorStateInfo.mxml -->

    <mx:TextInput id="newLabel" text="{data.label}" />
    <mx:TextInput id="newData" text="{data.data}" />
    <mx:TextInput id="newWebPage" text="{data.webPage}" />
</mx:VBox>
```

You define an item editor that contains three TextInput controls that let the user edit the state name, capital, or web address. This item editor returns three values, so you write an event listener for the itemEditEnd event to write the values to the data provider of the List control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\MainListStateRendererEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="700" width="700">

    <mx:Script>
        <![CDATA[

            import mx.events.ListEvent;
            import myComponents.EditorStateInfo;

            // Define the event listener.
          public function processData(event:ListEvent):void {

                // Disable copying data back to the control.
                event.preventDefault();

                // Get new label from editor.
                myList.editedItemRenderer.data.label =
EditorStateInfo(event.currentTarget.itemEditorInstance).newLabel.text;

                // Get new data from editor.
                myList.editedItemRenderer.data.data =
EditorStateInfo(event.currentTarget.itemEditorInstance).newData.text;

                // Get new webPage from editor.
                myList.editedItemRenderer.data.webPage =
EditorStateInfo(event.currentTarget.itemEditorInstance).newWebPage.text;

                // Close the cell editor.
                myList.destroyItemEditor();

                // Notify the list control to update its display.
        myList.dataProvider.notifyItemUpdate(myList.editedItemRenderer);
            }
        ]]>
    </mx:Script>

    <mx:List id="myList"
        height="180" width="250"
        editable="true"
        itemRenderer="myComponents.RendererState"
        itemEditor="myComponents.EditorStateInfo"
        variableRowHeight="true"
        backgroundColor="white"
```

```
        itemEditEnd="processData(event);">
        <mx:dataProvider>
            <mx:Object label="Alaska"
                data="Juneau"
                webPage="http://www.state.ak.us/"/>
            <mx:Object label="Alabama"
                data="Montgomery"
                webPage="http://www.alabama.gov/" />
            <mx:Object label="Arkansas"
                data="Little Rock"
                webPage="http://www.state.ar.us/"/>
        </mx:dataProvider>
    </mx:List>
</mx:Application>
```

This example uses the RendererState.mxml renderer. You can see that renderer in the section "Example: Using an item renderer with a List control" on page 897.

Using a DateField or ComboBox control as a drop-in item editor

You can use a DateField or ComboBox control as a drop-in item editor with the List control. However, when the data provider is a collection of Objects, you have to set the `labelField` property of the List control to the name of the field in the data provider modified by the DateField or ComboBox control, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\list\ListEditorDateField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.collections.*;
            import mx.controls.DateField;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var catalog:ArrayCollection = new ArrayCollection([
                {confirmed: new Date(), Location: "Spain"},
                {confirmed: new Date(2006,0,15), Location: "Italy"},
                {confirmed: new Date(2004,9,24), Location: "Bora Bora"},
                {confirmed: new Date(), Location: "Vietnam"}
            ]);
        ]]>
    </mx:Script>

    <mx:List id="myList"
        width="300" height="300"
        rowHeight="50"
        dataProvider="{catalog}"
        editable="true"
        labelField="confirmed"
        itemEditor="mx.controls.DateField"
        editorDataField="selectedDate"/>
</mx:Application>
```

In this example, you specify `"confirmed"` as the value of the `labelField` property to specify that the DateField control modifies that field of the data provider.

Example: Using a custom item editor with a Tree control

In a Tree control, you often display a single label for each node in the tree. However, the data provider for each node may contain additional data that is normally hidden from view.

In this example, you use the Tree control to display contact information for different companies. The Tree control displays the company name as a branch node, and different department names within the company as leaf nodes. Selecting any node opens a custom item editor that lets you modify the phone number or contact status of the company or for any department in the company.

The top node in the Tree control is not editable. Therefore, this example uses the `itemEditBeginning` event to determine if the user selects the top node. If selected, the event listener for the `itemEditBeginning` event prevents editing for occurring, as the following example shows:

```
<?xml version="1.0"?>
<!-- itemRenderers\tree\MainTreeEditor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600" height="600">

    <mx:Script>
        <![CDATA[

        import mx.events.ListEvent;
        import myComponents.TreeEditor;

        private var contacts1:Object =
          {label: "top", children: [
           {label: "Acme", status: true, phone: "243-333-5555", children: [
             {label: "Sales", status: true, phone: "561-256-5555"},
             {label: "Support", status: false, phone: "871-256-5555"}
           ]},
           {label: "Ace", status: true, phone: "444-333-5555", children: [
             {label: "Sales", status: true, phone: "232-898-5555"},
             {label: "Support", status: false, phone: "977-296-5555"},
           ]},
           {label: "Platinum", status: false, phone: "521-256-5555"}
        ]};

        private function initCatalog(cat:Object):void {
            myTree.dataProvider = cat;
        }

        // Define the event listener for the itemEditBeginning event
        // to disable editing when the user selects
        // the top node in the tree.
        private function disableEditing(event:ListEvent):void {
            if(event.rowIndex==0) {
                event.preventDefault();
            }
        }

        // Define the event listener for the itemEditEnd event
        // to copy the updated data back to the data provider
        // of the Tree control.
        public function processData(event:ListEvent):void {

            // Disable copying data back to the control.
            event.preventDefault();
```

```
            // Get new phone number from editor.
            myTree.editedItemRenderer.data.phone =
    TreeEditor(event.currentTarget.itemEditorInstance).contactPhone.text;

            // Get new status from editor.
            myTree.editedItemRenderer.data.status =
    TreeEditor(event.currentTarget.itemEditorInstance).confirmed.selected;

            // Close the cell editor.
            myTree.destroyItemEditor();

            // Notify the list control to update its display.
    myTree.dataProvider.notifyItemUpdate(myTree.editedItemRenderer);
        }
        ]]>
    </mx:Script>

    <mx:Tree id="myTree"
        width="400" height="400"
        editable="true"
        itemEditor="myComponents.TreeEditor"
        editorHeightOffset="75" editorWidthOffset="-100"
        editorXOffset="40" editorYOffset="30"
        creationComplete="initCatalog(contacts1);"
        itemEditBeginning="disableEditing(event);"
        itemEditEnd="processData(event);"/>
</mx:Application>
```

You specify the custom item editor using the `itemEditor` property of a tree control. You also use the `editorHeightOffset`, `editorWidthOffset`, `editorXOffset`, and `editorYOffset` properties to position the item editor.

The following item editor, defined in the file TreeEditor.mxml, lets you edit the data associated with each item in a Tree control:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- itemRenderers/tree/myComponents/TreeEditor.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Define variables for the new data.
            public var newPhone:String;
            public var newConfirmed:Boolean;

        ]]>
    </mx:Script>

    <!-- Display item label.-->
    <mx:Label text="{data.label}"/>

    <!-- Display the text 'Phone:' and let the user edit it.-->
    <mx:HBox>
        <mx:Text text="Phone:"/>
        <mx:TextInput id="contactPhone"
            width="150"
            text="{data.phone}"
            change="newPhone=contactPhone.text;"/>
    </mx:HBox>

    <!-- Display the status using a CheckBox control
        and let the user edit it.-->
    <mx:CheckBox id="confirmed"
        label="Confirmed"
        selected="{data.status}"
        click="newConfirmed=confirmed.selected;"/>
</mx:VBox>
```

CHAPTER 23

Using ToolTips

23

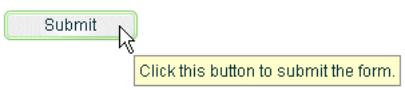
Adobe Flex ToolTips are a flexible method of providing helpful information to your users. When a user moves the mouse pointer over graphical components, ToolTips pop up and text appears. You can use ToolTips to guide users through working with your application or customize them to provide additional functionality. This topic describes how to use ToolTips in your Flex applications.

Contents

About ToolTips

ToolTips are a standard feature of many desktop applications. They make the application easier to use by displaying messages when the user moves the mouse pointer over an onscreen element, such as a Button control.

The following image shows ToolTip text that appears when the user hovers the mouse pointer over a button:



When the user moves the mouse pointer away from the component or clicks the mouse button, the ToolTip disappears. If the mouse pointer remains over the component, the ToolTip eventually disappears. The default behavior is to display only one ToolTip at a time.

You can set the time it takes for the ToolTip to appear when a user moves the mouse pointer over the component, and set the amount of time it takes for the ToolTip to disappear.

If you define a ToolTip on a container, the ToolTipManager displays the parent's ToolTip for the children if the child does not have one.

Flex ToolTips support style sheets and the dynamic loading of ToolTip text. ToolTip text does not support embedded HTML. For more information on using style sheets and dynamic loading of text, see "Setting styles in ToolTips" on page 946 and "Using dynamic ToolTip text" on page 956.

Some Flex components have their own ToolTip-like "data tips." These components include the List control, most chart controls, and DataGrid control. For more information, see that component's documentation.

ToolTips are the basis for the accessibility implementation within Flex. All controls that support accessibility do so by making their ToolTip text readable by screen readers such as JAWS. For more information on accessibility, see Chapter 36, "Creating Accessible Applications," on page 1219.

Creating ToolTips

Every visual Flex component that extends the UIComponent class (which implements the IToolTipManagerClient interface) supports a `toolTip` property. This property is inherited from the UIComponent class. You set the value of the `toolTip` property to a text string and, when the mouse pointer hovers over that component, the text string appears. The following example sets the `toolTip` property text for a Button control:

```
<?xml version="1.0"?>
<!-- tooltips/BasicToolTip.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Button id="b1" label="Click Me" toolTip="This Button does nothing."/>

</mx:Application>
```

To set the value of a ToolTip in ActionScript, use the `toolTip` property of the component. The following example creates a new Button and sets the `toolTip` property of a Button control:

```
<?xml version="1.0"?>
<!-- tooltips/BasicToolTipActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     public function createNewButton(event:MouseEvent):void {
        var myButton:Button = new Button();
        myButton.label = "Create Another Button";
        myButton.toolTip = "Click this new button to create another
button.";
        myButton.addEventListener(MouseEvent.CLICK, createNewButton);
        addChild(myButton);
     }
  ]]></mx:Script>
  <mx:Button id="b1" label="Create Another Button"
click="createNewButton(event);"/>
</mx:Application>
```

If you do not define a ToolTip on the child of a container, the ToolTipManager displays the parent's ToolTip. For example, if you add a Button control to a Panel container that has a ToolTip, the user sees the Panel container's ToolTip text when the user moves the mouse pointer over the Panel. When the user moves the mouse pointer over the Button control, the Panel's ToolTip continues to be displayed. You can override the container's ToolTip text by setting the value of the child's `toolTip` property.

The following example shows the inheritance of ToolTip text:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipInheritance.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:VBox toolTip="VBOX">
     <mx:Button id="b1" label="Button 1" toolTip="BUTTON"/>
     <mx:Button id="b2" label="Button 2"/>
  </mx:VBox>

</mx:Application>
```

When the mouse pointer is over button b1, the ToolTip displays BUTTON. When the mouse pointer is over button b2 or anywhere in the VBox container except over button b1, the ToolTip displays VBOX.

The TabNavigator container uses the ToolTips that are on its children. If you add a ToolTip to a child view of a TabNavigator container, the ToolTip appears when the mouse is over the tab for that view, but not when the mouse is over the view itself. If you add a ToolTip to the TabNavigator container, then the ToolTip appears when the mouse is over either the tab or the view, unless the ToolTip is overridden by the tab or the view. ToolTips in the following controls also behave this way:

- Accordion
- ButtonBar
- LinkBar
- TabBar
- ToggleButtonBar

There is no limit to the size of the ToolTip text, although long messages can be difficult to read. When the ToolTip text reaches the width of the ToolTip box, the text wraps to the next line. You can add line breaks in ToolTip text. In ActionScript, you use the \n escaped newline character. In MXML tags, you use the
 XML entity.

The following examples show using the \n escaped newline character and the
 entity:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipNewlines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="doSomething(event)">
  <mx:Script><![CDATA[
     public function doSomething(event:Event):void {
        // Use the \n to force a line break in ActionScript.
        b1.toolTip = "Click this button \n to clear the form.";
     }
  ]]></mx:Script>
  <mx:Button id="b1" label="Clear" width="100"/>
  <!-- Use &#13; to force a line break in MXML tags. -->
  <mx:Button id="b2" label="Submit" width="100" toolTip="Click this button
&#13; to submit the form."/>
</mx:Application>
```

You also have some flexibility in formatting the text of the ToolTip. You can apply styles and change other settings for all ToolTips in your application by using the ToolTip CSS type selector. The following sections describe how to set styles on the ToolTip text and box.

Setting styles in ToolTips

You can change the appearance of ToolTip text and the ToolTip box by using Cascading Style Sheets (CSS) syntax or the mx.styles.StyleManager class. Changes to ToolTip styles apply to all ToolTips in the current application.

The default styles for ToolTips are defined by the ToolTip type selector in the defaults.css file in the framework.swc file. You can use a type selector in the `<mx:Style>` tag to override the default styles of your ToolTips. The following example sets the styles of the ToolTip type selector using CSS syntax:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     ToolTip {
        fontFamily: "Arial";
        fontSize: 19;
        fontStyle: "italic";
        color: #FF6699;
        backgroundColor: #33CC99;
     }
  </mx:Style>

  <mx:Button id="b1" label="Click Me" toolTip="This Button does nothing."/>

</mx:Application>
```

To use the StyleManager class to set ToolTip styles, apply a style to the ToolTip type selector with the `setStyle()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipStyleManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="setToolTipStyle()">

  <mx:Script><![CDATA[
     import mx.styles.StyleManager;
     private function setToolTipStyle():void {

StyleManager.getStyleDeclaration("ToolTip").setStyle("fontStyle","italic");

StyleManager.getStyleDeclaration("ToolTip").setStyle("fontSize","19");

StyleManager.getStyleDeclaration("ToolTip").setStyle("fontFamily","Arial");

StyleManager.getStyleDeclaration("ToolTip").setStyle("color","#FF6699");

StyleManager.getStyleDeclaration("ToolTip").setStyle("backgroundColor","#33
CC99");
     }
]]></mx:Script>

  <mx:Button id="b1" label="Click Me" toolTip="This Button does nothing."/>

</mx:Application>
```

ToolTips use inheritable styles that you set globally. For example, you can set the `fontWeight` of ToolTips with the StyleManager by setting it on the global selector, as the following example shows:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipGlobalStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="setToolTipStyle()">

  <mx:Script><![CDATA[
     import mx.styles.StyleManager;
     private function setToolTipStyle():void {

StyleManager.getStyleDeclaration("global").setStyle("fontWeight","bold");
     }
]]></mx:Script>

  <mx:Button id="b1" label="Click Me" toolTip="This Button does nothing."/>

</mx:Application>
```

If you set the `fontWeight` property on the global selector, your change affects the text of many controls in addition to ToolTips, so be careful when using the global selector.

You can reskin a ToolTip with style properties and programmatic skins. For more information, see "Reskinning ToolTips" on page 847.

For a complete list of styles supported by ToolTips, see *Adobe Flex 2 Language Reference*. For more information on using styles, see Chapter 18, "Using Styles and Themes," on page 697.

You can reskin ToolTip controls to give them an entirely new appearance. You can do this by using the ToolTipBorder programmatic skin or reskin them graphically. For more information on skinning ToolTips, see Chapter 20, "Using Skins," on page 805. For an example of programmatically reskinning a ToolTip control, see "Reskinning ToolTips" on page 847.

Setting the width of ToolTips

You can set the width of the ToolTip box by changing the `maxWidth` property of the mx.controls.ToolTip class. This property is static so when you set it, you are setting it for all ToolTip boxes and you cannot set it on an instance of a ToolTip.

The `maxWidth` property specifies the maximum width in pixels for new ToolTips boxes. For example, the following line changes the maximum width of the ToolTip box to 100 pixels:

```
<?xml version="1.0"?>
<!-- tooltips/SetMaxWidth.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.controls.ToolTip;

     public function init():void {
        ToolTip.maxWidth = 100;
     }

     public function createNewButton(event:MouseEvent):void {
        var myButton:Button = new Button();
        myButton.label = "Create Another Button";
        myButton.toolTip = "Click this new button to create another
button.";
        myButton.addEventListener(MouseEvent.CLICK, createNewButton);
        addChild(myButton);

     }
  ]]></mx:Script>
  <mx:Button id="b1" label="Create Another Button"
click="createNewButton(event);" toolTip="Click this button to create a new
one."/>
</mx:Application>
```

Flex wraps the text of a ToolTip onto multiple lines to ensure that the width does not exceed this value. If the text in the ToolTip box is not as wide as the `maxWidth` property, Flex creates a box only wide enough for the text to fit.

The default value of `maxWidth` is `300`. If the value of `maxWidth` exceeds the width of the application, Flex clips the text in the ToolTip box.

Using ToolTip events

ToolTips trigger many events during their life cycle. These events are of type ToolTipEvent.

In addition to the `type` and `target` properties, the ToolTipEvent object references the ToolTip in its `toolTip` property. With a reference to the ToolTip, you can then access the ToolTip text with the `text` property.

To use events of type ToolTipEvent in your `<mx:Script>` blocks, you must import mx.events.ToolTipEvent class.

The following example plays a sound in response to the TOOL_TIP_SHOW event:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipsWithSoundEffects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="init()">
  <mx:Script><![CDATA[
     import mx.events.ToolTipEvent;
     import flash.media.Sound;

     [Embed(source="../assets/sound1.mp3")]
     private var beepSound:Class;
     private var myClip:Sound;

     public function playSound():void {
        myClip.play();
     }
     private function myListener(event:ToolTipEvent):void {
        playSound();
     }
     private function init():void {
        myLabel.addEventListener(ToolTipEvent.TOOL_TIP_SHOW, myListener);
        myClip = new beepSound();
     }
  ]]></mx:Script>
  <mx:Label id="myLabel" toolTip="ToolTip" text="Mouse Over Me"/>
</mx:Application>
```

Using ToolTips with NavBar controls

NavBar and TabBar subclasses (such as ButtonBar, LinkBar, and ToggleButtonBar) support ToolTips in their data providers. The data provider array can contain a toolTip field that specifies the toolTip for the navigation items.

The following example creates ToolTips for each of the navigation items:

```
<?xml version="1.0"?>
<!-- tooltips/NavItemToolTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:ButtonBar>
     <mx:Object label="OK" toolTip="OK tip"/>
     <mx:Object label="Cancel" toolTip="Cancel tip"/>
  </mx:ButtonBar>
</mx:Application>
```

You can use ToolTips on child elements in a component's data provider. The component recognizes those ToolTips and displays them accordingly. In the following example, the ToolTips are propagated up to the LinkBar:

```
<?xml version="1.0"?>
<!-- tooltips/DataProviderToolTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:VBox>
     <!-- Create a LinkBar control to navigate the ViewStack. -->
     <mx:LinkBar dataProvider="{vs1}" borderStyle="solid"/>

     <!-- Define the ViewStack and the three child containers. -->
     <mx:ViewStack id="vs1" borderStyle="solid" width="100%" height="150">
        <mx:Canvas id="search" label="Search" toolTip="Search Screen">
           <mx:Label text="Search Screen"/>
        </mx:Canvas>
        <mx:Canvas id="custInfo" label="Customer"
           toolTip="Customer Info Screen">
           <mx:Label text="Customer Info"/>
        </mx:Canvas>
        <mx:Canvas id="accountInfo" label="Account"
           toolTip="Account Info Screen">
           <mx:Label text="Account Info"/>
        </mx:Canvas>
     </mx:ViewStack>
  </mx:VBox>
</mx:Application>
```

You can also set the value of the NavBar's `toolTipField` property to point to the field in the data provider that provides a ToolTip. The data provider in the following example defines ToolTips in the myToolTip field:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipFieldExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:ButtonBar toolTipField="myToolTip">
     <mx:Object label="OK" myToolTip="OK tip"/>
     <mx:Object label="Cancel" myToolTip="Cancel tip"/>
  </mx:ButtonBar>
</mx:Application>
```

Using the ToolTip Manager

The ToolTipManager class lets you set basic ToolTip functionality, such as display delay and the disabling of ToolTips. It is located in the mx.managers package. You must import this class when using the ToolTipManager. The ToolTipManager class also contains a reference to the current ToolTip in its `currentToolTip` property.

This section describes how to use the ToolTip Manager.

Enabling and disabling ToolTips

You can enable and disable ToolTips in your Flex applications. When you disable ToolTips, no ToolTip box appears when the user moves the mouse pointer over a visible component, regardless of whether that component's `toolTip` property is set.

You use the `enabled` property of the ToolTipManager to enable or disable ToolTips. You set this property to `true` to enable ToolTips or `false` to disable ToolTips. The default value is `true`.

The following example toggles ToolTips on and off when the user clicks the Toggle ToolTips button:

```
<?xml version="1.0"?>
<!-- tooltips/ToggleToolTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script><![CDATA[
     import mx.managers.ToolTipManager;

     private function toggleToolTips():void {
        if (ToolTipManager.enabled) {
           ToolTipManager.enabled = false;
        } else {
           ToolTipManager.enabled = true;
        }
     }
  ]]></mx:Script>

  <mx:Button label="Toggle ToolTips" width="100" click="toggleToolTips()"
toolTip="Click me to enable/disable tooltips."/>

</mx:Application>
```

Setting delay times

A *delay time* is a measurement of time that passes before something takes place. For example, after the user moves the mouse pointer over a component, there is a brief delay before the ToolTip appears. This gives someone who is not looking for ToolTip text enough time to move the mouse pointer away before seeing the pop-up.

The ToolTipManager lets you set the length of time that passes before the ToolTip box appears, and the length of time that a ToolTip remains on the screen when a mouse pointer hovers over the component. You can also set the delay between ToolTips.

You set the value of the ToolTipManager `showDelay`, `hideDelay`, and `scrubDelay` properties in your ActionScript code blocks. The following table describes the time delay properties of the ToolTipManager:

Property	Description
`showDelay`	The length of time, in milliseconds, that Flex waits before displaying the ToolTip box when a user moves the mouse pointer over a component that has a ToolTip. To make the ToolTip appear instantly, set the `showDelay` property to 0. The default value is 500 milliseconds, or half of a second.
`hideDelay`	The length of time, in milliseconds, that Flex waits to hide the ToolTip box after it appears. After Flex hides a ToolTip box, the user must move the mouse pointer off the component and back onto it to see the ToolTip box again. If you set the `hideDelay` property to 0, Flex does not display the ToolTip. Adobe recommends that you use the default value of 10,000 milliseconds, or 10 seconds. If you set the `hideDelay` property to `Infinity`, Flex does not hide the ToolTip until the user triggers an event (such as moving the mouse pointer off the component). The following example sets the `hideDelay` property to `Infinity`: `import mx.managers.ToolTipManager;` `ToolTipManager.hideDelay = Infinity;`
`scrubDelay`	The length of time, in milliseconds, that a user can take when moving the mouse between controls before Flex again waits for the duration of `showDelay` to display a ToolTip box. This setting is useful if the user moves quickly from one control to another; after displaying the first ToolTip, Flex displays the others immediately rather than waiting for the duration of the `showDelay` setting. The shorter the setting for `scrubDelay`, the more likely that the user must wait for an amount of time specified by the `showDelay` property in order to see the next ToolTip. A good use of this property is if you have several buttons on a toolbar, and the user will quickly scan across them to see brief descriptions of their functionality. The default value is 100.

The following example uses the Application control's `initialize` event to set the starting values for the ToolTipManager:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipTiming.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initApp();">
  <mx:Script><![CDATA[
     import mx.managers.ToolTipManager;
     private function initApp():void {
        ToolTipManager.enabled = true;// Optional. Default value is true.
        ToolTipManager.showDelay = 0;// Display immediately.
        ToolTipManager.hideDelay = 3000; // Hide after 3 seconds.
     }
  ]]></mx:Script>

  <mx:Button label="Click Me" toolTip="Click this Button to do something"/>
  <mx:Button label="Click Me" toolTip="Click this Button to do something
else"/>
  <mx:Button label="Click Me" toolTip="Click this Button to do another
thing"/>
  <mx:Button label="Click Me" toolTip="Click this Button to do the same
thing"/>

</mx:Application>
```

Using effects with ToolTips

You can use a custom effect or one of the standard Flex effects with ToolTips. You set the `showEffect` or `hideEffect` property of the ToolTipManager to define the effect that is triggered whenever a ToolTip is displayed or is hidden.

The following example uses the Fade effect so that ToolTips fade in when the user moves the mouse pointer over a component with a ToolTip:

```
<?xml version="1.0"?>
<!-- tooltips/FadeInToolTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600" initialize="app_init();">

  <mx:Script><![CDATA[
     import mx.managers.ToolTipManager;
     public function app_init():void {
        ToolTipManager.showEffect = fadeIn;
     }
  ]]></mx:Script>

  <mx:Fade id="fadeIn" alphaFrom="0" alphaTo="1" duration="1000"/>

  <mx:Button id="b1" label="Click me" toolTip="This is a ToolTip."/>

</mx:Application>
```

By default, the font used in this example does not fade. You must use an embedded font to achieve the embed effect. For more information on using embedded fonts, see "Using embedded fonts" on page 767.

For more information about using effects and defining custom effects, see Chapter 17, "Using Behaviors," on page 649.

Using dynamic ToolTip text

You can use ToolTips for more than just displaying static help text to the user. You can also bind the ToolTip text to data or component text. This lets you use ToolTips as a part of the user interface, showing drill-down information, query results, or more helpful text that is customized to the user experience.

You bind the value of the ToolTip text to the value of another control's text using curly braces ({ }) syntax.

The following example inserts the value of the TextInput control into the Button control's ToolTip text when the user moves the mouse pointer over the Button control:

```
<?xml version="1.0"?>
<!-- tooltips/BoundToolTipText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:TextInput id="txtTo" width="300"/>
  <mx:Button label="Send" toolTip="Send e-mail to {txtTo.text}"/>

</mx:Application>
```

In this example, if the user enters **fred@fred.com** in the TextInput box, and then moves the mouse pointer over the button, Flex displays the message "Send e-mail to fred@fred.com" in the ToolTip box.

Another approach to creating dynamic text for ToolTips is to intercept the ToolTip in its `toolTipShow` event handler and change the value of its `text` property. The following example registers the `myToolTipChanger()` method as a listener for the Button control's `toolTipShow` event. The code in that method changes the value of the `ToolTipManager.currentToolTip.text` property to a value that is not known until run time.

```
<?xml version="1.0"?>
<!-- tooltips/DynamicToolTipText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initApp()">
  <mx:Script><![CDATA[
     import mx.managers.ToolTipManager;
     import mx.controls.ToolTip;
     import mx.events.ToolTipEvent;

     public function initApp():void {
        b1.addEventListener(ToolTipEvent.TOOL_TIP_SHOW, myToolTipChanger)
     }
     public function myToolTipChanger(event:ToolTipEvent):void {
        // Append ?myName=fred to your request string or pass the value of
        // myName in to your application some other way.
        ToolTipManager.currentToolTip.text = "Click the button, " +
           Application.application.parameters.myName;
     }
  ]]>   </mx:Script>
  <mx:Button id="b1" label="Click Me" toolTip="Click the button"/>
</mx:Application>
```

You can only create text for the current ToolTip in the `toolTipShow` event handler if the object had a ToolTip in the first place. For example, if the button in the previous example did not set a value for the `toolTip` property, no ToolTip would appear even after the `myToolTipChanger()` method was called.

For information about using the `Application.application.parameters` object, see "Using the Application.application.parameters object" on page 1186.

Creating custom ToolTips

The ToolTipManager has two methods that let you programmatically use ToolTips. These methods are `createToolTip()` and `destroyToolTip()`, which you use to create and destroy new ToolTip objects. When you create a ToolTip object, you can customize it as you would any object, with access to its properties, styles, events, and effects.

This `createToolTip()` method has the following signature:

```
createToolTip(text:String, x:Number, y:Number, errorTipBorderStyle:String,
  context:IUIComponent):IToolTip
```

The `createToolTip()` method returns an IToolTip, an interface that the ToolTip class implements. You must often cast the return value of this method to a ToolTip, although it is more efficient if you do not do this. To cast, you can do one of the following:

- Use the `as` keyword, as the following example shows:

  ```
  myTip = ToolTipManager.createToolTip(s,10,10) as ToolTip;
  ```

- Use the *`type(object)`* casting syntax, as the following example shows:

  ```
  myTip = ToolTip(ToolTipManager.createToolTip(s,10,10));
  ```

These methods of casting differ only in the way they behave when a cast fails.

The *`text`* parameter defines the contents of the ToolTip. The *`x`* and *`y`* parameters define the *x* and *y* coordinates of the ToolTip, relative to the application container. The *`errorTipBorderStyle`* property sets the location of the pointer on the error tip. This parameter is optional. If you specify this value in the `createToolTip()` method, Flex styles the ToolTip as an error tip. The following example shows the valid values and their resulting locations on the error tip:

The *`context`* parameter is not currently used.

For more information about using error tips, see "Using error tips" on page 962.

Flex displays the ToolTip until you destroy it. In general, you should not display more than one ToolTip box at a time, because it is confusing to the user.

You can use the `destroyToolTip()` method to destroy the specified ToolTip object. The `destroyToolTip()` method has the following signature:

```
destroyToolTip(toolTip:IToolTip):void
```

The *toolTip* parameter is the ToolTip object that you want to destroy. This is the object returned by the `createToolTip()` method.

The following example creates a custom ToolTip when you move the mouse over a Panel container that contains three Button controls. Each Button control has its own ToolTip that appears when you mouse over that particular control. The big ToolTip only disappears when you move the mouse away from the Panel container.

```
<?xml version="1.0"?>
<!-- tooltips/CreatingToolTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.managers.ToolTipManager;
  import mx.controls.ToolTip;

  public var myTip:ToolTip;

  private function createBigTip():void {
     var s:String = "These buttons let you save, exit, or continue with the
current operation."
     myTip = ToolTipManager.createToolTip(s,10,10) as ToolTip;
     myTip.setStyle("backgroundColor",0xFFCC00);
     myTip.width = 150;
     myTip.height = 200;
  }

  private function destroyBigTip():void {
     ToolTipManager.destroyToolTip(myTip);
  }
  ]]></mx:Script>
  <mx:Panel rollOver="createBigTip()" rollOut="destroyBigTip()">
     <mx:Button label="OK" toolTip="Save your changes and exit."/>
     <mx:Button label="Apply" toolTip="Apply changes and continue."/>
     <mx:Button label="Cancel" toolTip="Cancel and exit."/>
  </mx:Panel>
</mx:Application>
```

You can also create a custom ToolTip by extending an existing control, such as a Panel or VBox container, and implementing the IToolTip interface. The following example uses a Panel container as the base for a new implementation of the IToolTipInterface:

```
<?xml version="1.0"?>
<!-- tooltips/ToolTipComponents/PanelToolTip.mxml -->
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml"
    implements="mx.core.IToolTip"
    width="200"
    alpha=".8"
    borderThickness="2"
    backgroundColor="0xCCCCCC"
    dropShadowEnabled="true"
    borderColor="black"
    borderStyle="solid"
>
    <mx:Script><![CDATA[
        [Bindable]
        public var bodyText:String = "heh";

        //  Implement required methods of the IToolTip interface; these
        //  methods are not used in this example, though.
        public var _text:String;

        public function get text():String {
            return _text;
        }
        public function set text(value:String):void {
        }
    ]]></mx:Script>

    <mx:Text text="{bodyText}" percentWidth="100"/>

</mx:Panel>
```

In your application, you can create a custom ToolTip by intercepting the `toolTipCreate` event handler of the target component. In the event handler, you instantiate the new ToolTip and set its properties. You then point the `toolTip` property of the ToolTipEvent object to the new ToolTip.

In the following example, the first two buttons in the application use the custom PanelToolTip in the CustomToolTips package. The third button uses a default ToolTip to show you how the two are different. To run this example, store the PanelToolTip.mxml file in a subdirectory named CustomToolTips.

```
<?xml version="1.0"?>
<!-- tooltips/MainCustomApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import ToolTipComponents.PanelToolTip;
        import mx.events.ToolTipEvent;

        private function createCustomTip(title:String, body:String,
event:ToolTipEvent):void {
           var ptt:PanelToolTip = new PanelToolTip();
           ptt.title = title;
           ptt.bodyText = body;
           event.toolTip = ptt;
        }
    ]]></mx:Script>

    <mx:Button id="b1"
        label="Delete"
        toolTip=" "
       toolTipCreate="createCustomTip('DELETE','Click this button to delete
the report.', event)"
    />

    <mx:Button id="b2"
        label="Generate"
        toolTip=" "
        toolTipCreate="createCustomTip('GENERATE','Click this button to
generate the report.', event)"
    />

    <mx:Button id="b3"
        label="Stop"
       toolTip="Click this button to stop the creation of the report. This
button uses a standard ToolTip style."
    />

</mx:Application>
```

Using error tips

Error tips are instances of the ToolTip class that get their styles from the `errorTip` class selector. They are most often seen when the Flex validation mechanism displays a warning when data is invalid. But you can use the definitions of the `errorTip` style and apply it to ToolTips to create a custom validation warning mechanism.

The styles of an error tip are defined in the defaults.css file, which is in the framework.swc file. It specifies the following default settings for `errorTip` (notice the period preceding `errorTip`, which indicates that it is a class selector):

```
.errorTip {
  color: #FFFFFF;
  fontSize: 9;
  fontWeight: "bold";
  shadowColor: #000000;
  borderColor: #CE2929;
  borderStyle: "errorTipRight";
  paddingBottom: 4;
  paddingLeft: 4;
  paddingRight: 4;
  paddingTop: 4;
}
```

You can customize the appearance of error tips by creating a new theme that overrides these styles, or by overriding the style properties in your application. For more information on creating themes, see "About themes" on page 756.

You can create ToolTips that look like validation error tips by applying the `errorTip` style to the ToolTip. The following example does not contain any validation logic, but shows you how to use the `errorTip` style to create ToolTips that look like validation error tips. When you run the example, press the Enter key after entering text into the TextInput controls.

```
<?xml version="1.0"?>
<!-- tooltips/ErrorTipStyle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="20">
  <mx:Script><![CDATA[
     import mx.controls.ToolTip;
     import mx.managers.ToolTipManager;

     private var errorTip:ToolTip;
     private var myError:String;

     private function validateEntry(type:String, event:Object):void {
        // NOTE: Validation logic would go here.
        switch(type) {
           case "ssn":
            myError="Use SSN format (NNN-NN-NNNN)";
            break;
           case "phone":
            myError="Use phone format (NNN-NNN-NNNN)";
            break;
        }
        // Use the target's x and y positions to set position of error tip.
        errorTip =
ToolTipManager.createToolTip(myError,event.currentTarget.x +
event.currentTarget.width,event.currentTarget.y) as ToolTip;

        // Apply the errorTip class selector.
        errorTip.setStyle("styleName", "errorTip");
     }
  ]]></mx:Script>

  <mx:TextInput id="ssn" enter="validateEntry('ssn',event)"/>
  <mx:TextInput id="phone" enter="validateEntry('phone',event)"/>
</mx:Application>
```

Another way to use error tips is to set the value of the `errorString` property of the component. This causes the ToolTipManager to create an instance of a ToolTip and apply the `errorTip` style to that ToolTip without requiring any coding on your part.

The following example shows how to set the value of the `errorString` property to create an error tip:

```
<?xml version="1.0"?>
<!-- tooltips/ErrorString.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="20">
  <mx:Script><![CDATA[
     import mx.controls.ToolTip;
     import mx.managers.ToolTipManager;

     private var errorTip:ToolTip;
     private var myError:String;

     private function validateEntry(type:String, event:Object):void {
        // NOTE: Validation logic would go here.
        switch(type) {
           case "ssn":
            myError="Use SSN format";
            break;
           case "phone":
            myError="Use phone format";
            break;
        }

        event.currentTarget.errorString = myError;

     }
  ]]></mx:Script>

  <mx:TextInput id="ssn" enter="validateEntry('ssn',event)"/>
  <mx:TextInput id="phone" enter="validateEntry('phone',event)"/>
</mx:Application>
```

You can also specify that the ToolTipManager creates an error tip when you call the `createToolTip()` method by specifying the value of the `errorTipBorderStyle` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- tooltips/CreatingErrorTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.managers.ToolTipManager;
  import mx.core.IToolTip;

  public var myTip:IToolTip;

  private function createBigTip(event:Event):void {
     var myError:String = "These buttons let you save, exit, or continue
with the current operation."
     myTip = ToolTipManager.createToolTip(myError, event.currentTarget.x +
event.currentTarget.width, event.currentTarget.y, "errorTipRight");
  }

  private function destroyBigTip():void {
     ToolTipManager.destroyToolTip(myTip);
  }

  ]]></mx:Script>
  <mx:Panel rollOver="createBigTip(event)" rollOut="destroyBigTip()">
     <mx:Button label="OK" toolTip="Save your changes and exit."/>
     <mx:Button label="Apply" toolTip="Apply changes and continue."/>
     <mx:Button label="Cancel" toolTip="Cancel and exit."/>
  </mx:Panel>
</mx:Application>
```

For more information on using validators, see Chapter 40, “Validating Data,” on page 1281.

CHAPTER 24

Using the Cursor Manager

24

The Adobe Flex Cursor Manager lets you control the cursor image in your Flex application. You can use the Cursor Manager to provide visual feedback to users to indicate when to wait for processing to complete, to indicate allowable actions, or to provide other types of feedback. The cursor image can be a JPEG, GIF, PNG, or SVG image, a Sprite object, or a SWF file.

This topic describes how to use the Cursor Manager to control the cursor image in your Flex applications.

Contents

About the Cursor Manager

By default, Flex uses the system cursor as the application cursor. You control the system cursor by using the settings of your operating system.

The Flex Cursor Manager lets you control the cursor image in your Flex application. For example, if your application performs processing that requires the user to wait until the processing completes, you can change the cursor so that it reflects the waiting period. In this case, you can change the cursor to an hourglass or other image.

You also can change the cursor to provide feedback to the user to indicate the actions that the user can perform. For example, you can use one cursor image to indicate that user input is enabled, and another to indicate that input is disabled.

You can use a JPEG, GIF, PNG, or SVG image, a Sprite object, or a SWF file as the cursor image.

Using the Cursor Manager

To use the Cursor Manager, you import the mx.managers.CursorManager class into your application, and then reference its properties and methods.

The Cursor Manager controls a prioritized list of cursors, where the cursor with the highest priority is currently visible. If the cursor list contains more than one cursor with the same priority, the Cursor Manager displays the most recently created cursor.

You create a new cursor, and set an optional priority for the cursor, by using the static `setCursor()` method of the CursorManager class. This method adds the new cursor to the cursor list. If the new cursor has the highest priority, it is displayed immediately. If the priority is lower than a cursor already in the list, it is not displayed until the cursor with the higher priority is removed.

To remove a cursor from the list, you use the static `removeCursor()` method. If the cursor is the currently displayed cursor, the Cursor Manager displays the next cursor in the list, if one exists. If the list ever becomes empty, the Cursor Manager displays the default system cursor.

The `setCursor()` method has the following signature:

```
public static setCursor(cursorClass:Class, priority:int = 2,
  xOffset:Number = 0, yOffset:Number = 0) : int
```

The following table describes the arguments for the `setCursor()` method:

Argument	Description	Req/ Opt
`cursorClass`	The class name of the cursor to display.	Required
`priority`	The priority level of the cursor. Valid values are `CursorManagerPriority.HIGH`, `CursorManagerPriority.MEDIUM`, and `CursorManagerPriority.LOW`. The default value is 2, corresponding to `CursorManagerPriority.MEDIUM`.	Optional
`xOffset`	The x offset of the cursor relative to the mouse pointer. The default value is 0.	Optional
`yOffset`	The y offset of the cursor relative to the mouse pointer. The default value is 0.	Optional

This method returns the ID of the new cursor. You pass the ID to the `removeCursor()` method to delete the cursor. This method has the following signature:

```
static removeCursor(cursorID:int):void
```

Creating and removing a cursor

The following example changes the cursor to a custom *wait* or *busy* cursor while a large image file loads. After the load completes, the application removes the busy cursor and returns the cursor to the system cursor.

NOTE

The Flex framework includes a default busy cursor. For information on using this cursor see "Using a busy cursor" on page 970.

```
<?xml version="1.0"?>
<!-- cursors\CursorManagerApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.CursorManager;
            import flash.events.*;

            // Define a variable to hold the cursor ID.
            private var cursorID:Number = 0;

            // Embed the cursor symbol.
            [Embed(source="assets/wait.jpg")]
            private var waitCursorSymbol:Class;

            // Define event listener to display the wait cursor
            // and to load the image.
            private function initImage(event:MouseEvent):void {
                // Set busy cursor.
                cursorID = CursorManager.setCursor(waitCursorSymbol);
                // Load large image.
                image1.load("assets/DSC00034.JPG");
            }

            // Define an event listener to remove the wait cursor.
            private function loadComplete(event:Event):void {
                CursorManager.removeCursor(cursorID);
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <!-- Image control to load the image. -->
        <mx:Image id="image1" complete="loadComplete(event);"/>

        <!-- Button triggers the load. -->
        <mx:Button id="myButton" label="Show" click="initImage(event);"/>
    </mx:VBox>
```

```
</mx:Application>
```

This example uses a JPEG image as the cursor image. You can also use a Sprite, or a PNG, GIF, SVG, or SWF file, as the following example shows:

```
[Embed(source="assets/wait.swf")]
var waitCursorSymbol:Class;
```

Or, you can reference a symbol from a SWF file, as the following example shows:

```
[Embed(source="assets/cursorList.swf", symbol="wait")]
var waitCursorSymbol:Class;
```

An advantage of using a SWF file is that you can create an animated cursor.

Using a busy cursor

Flex defines a default busy cursor that you can use to indicate that your application is processing, and that users should wait until that processing completes before the application will respond to inputs. The default busy cursor is an animated clock.

You can control a busy cursor in several ways:

- You can use CursorManager methods to set and remove the busy cursor.
- You can use the `showBusyCursor` property of the SWFLoader, WebService, HttpService, and RemoteObject classes to automatically display the busy cursor.

Setting a busy cursor

The following static Cursor Manager methods control the busy cursor:

Method	Description
`setBusyCursor()`	Displays the busy cursor.
`removeBusyCursor()`	Removes the busy cursor from the cursor list. If other busy cursor requests are still active in the cursor list, which means that you called the `setBusyCursor()` method more than once, a busy cursor does not disappear until you remove all busy cursors from the list.

You can modify the example in "Creating and removing a cursor" on page 969 to use the default busy cursor, as the following example shows:

```
<?xml version="1.0"?>
<!-- cursors\DefBusyCursorApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.CursorManager;
            import flash.events.*;

            private function initImage(event:MouseEvent):void {
                CursorManager.setBusyCursor();
                image1.load("assets/DSC00034.JPG");
            }

            private function loadComplete(event:Event):void {
                CursorManager.removeBusyCursor();
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <!-- Image control to load the image. -->
        <mx:Image id="image1" complete="loadComplete(event);"/>

        <!-- Button triggers the load. -->
        <mx:Button id="myButton" label="Show" click="initImage(event);"/>
    </mx:VBox>
</mx:Application>
```

Setting the busy cursor does not prevent a user from interacting with your application; a user can still enter text or select buttons. However, all containers support the `enabled` property. By default, this property is set to `true` to enable user interaction with the container and with the container's children. If you set the `enabled` property to `false` when you display a busy cursor, Flex dims the color of the container and of all of its children, and blocks user input to the container and to all of its children.

You can also disable user interaction for the entire application by setting the `Application.application.enabled` property to `false`.

The busy cursor has a priority of `CursorManagerPriority.LOW`. Therefore, if the cursor list contains a cursor with a higher priority, the busy cursor does not appear until you remove the higher-priority cursor. To create a default busy cursor at a higher priority level, use the `setCursor()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- cursors\ShowBusyCursorAppHighP.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.managers.CursorManager;
            import mx.managers.CursorManagerPriority;
            import flash.events.*;

            // Define a variable to hold the cursor ID.
            private var cursorID:Number = 0;

            // Define event listener to display the busy cursor
            // and to load the image.
            private function initImage(event:MouseEvent):void {
                // Set busy cursor.
                cursorID=CursorManager.setCursor(
StyleManager.getStyleDeclaration("CursorManager").getStyle("busyCursor"),
CursorManagerPriority.HIGH);
                // Load large image.
                image1.load("assets/DSC00034.JPG");
            }

            // Define an event listener to remove the wait cursor.
            private function loadComplete(event:Event):void {
                CursorManager.removeCursor(cursorID);
            }
        ]]>
    </mx:Script>

    <mx:VBox>
        <!-- Image control to load the image. -->
        <mx:Image id="image1" complete="loadComplete(event);"/>

        <!-- Button triggers the load. -->
        <mx:Button id="myButton" label="Show" click="initImage(event);"/>
    </mx:VBox>
</mx:Application>
cursorID=CursorManager.setCursor(StyleManager.getStyleDeclaration
  ("CursorManager").getStyle("busyCursor"), CursorManagerPriority.HIGH);
```

This statement uses the static `getStyleDeclaration()` method of the StyleManager class to get the CSStyleDeclaration object for the Cursor Manager, and uses this object's `getStyle()` method to get the busy cursor, which it sets as a high priority cursor.

When you use this technique, you must also use the cursor ID in the `removeCursor()` method to remove the busy cursor.

Using the showBusyCursor property

The SWFLoader control, and the `<mx:WebService>`, `<mx:HttpService>`, and `<mx:RemoteObject>` tags have a `showBusyCursor` property that automatically displays the default busy cursor until the class completes loading data. The default value is `false` for the SWFLoader control, and for the `<mx:WebService>`, `<mx:HttpService>`, and `<mx:RemoteObject>` tags.

For the SWFLoader control, if you set the `showBusyCursor` property to `true`, Flex displays the busy cursor when the first `progress` event of the control is triggered, and hides the busy cursor when the `complete` event is triggered. The following example shows how you can simplify the example in the section "Setting a busy cursor" on page 970 by using the `showBusyCursor` property:

```
<?xml version="1.0"?>
<!-- cursors\ShowBusyCursorApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <!-- SWFLoader control to load the image. -->
        <mx:SWFLoader id="image1" showBusyCursor="true"/>

        <!-- Button triggers the load. -->
        <mx:Button id="myButton" label="Show"
            click="image1.load('assets/DSC00034.jpg');"/>
    </mx:VBox>
</mx:Application>
```

CHAPTER 25

Localizing Flex Applications

25

Adobe Flex provides localization support for Flex framework and custom components and classes.

Contents

About localized Flex applications

Using Flex, you can write localized applications and access localized components. Flex supports static inclusion of localized resources, but not dynamic retrieval of resources at run time.

About resource bundles and properties files

Resource bundles, or properties files, map keys and values by using simple "key=value" syntax. This is similar to Java properties files; the primary difference from the Java properties format is that you store the Flex properties files as UTF-8 if they contain non-ASCII characters.

The Flex compilers find the properties files the same way that the compilers find other source files. You put the properties files in directories that the compiler searches.

You can access the values in the resource bundles in the following ways:

MXML Use the `@Resource` directive inside your MXML tags.

ActionScript Use the `[ResourceBundle]` metadata tag to create an instance of the ResourceBundle class in ActionScript.

In some cases, you must also specify the name of the resource bundle to access the data inside it. The resource bundle in your code uses the same name as the properties file.

The localization workflow

To localize an application, you use the ResourceBundle API. You do this by adding `[ResourceBundle]` metadata tags in ActionScript or `@Resource` calls in MXML, creating localized properties files and localization classes and put them into a locale directory, and using the mxmlc compiler to compile the main application.

Optionally, you can package libraries of localized properties files and ActionScript classes in a SWC file that you create with the compc compiler. However, a resource bundle SWC file is necessary only when you are creating a library rather than an application.

For more information about the mxmlc and compc utilities, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Creating a localized application

This section begins with a simple sample localized application. It describes the key aspects in creating a localized application.

To build a simple localized application:

1. Create a locale/en_US/RegistrationForm.properties file that contains the following text:

   ```
   registration_title=Registration
   submit=Submit Form
   personname=Name
   street_address=Street Address
   city=City
   state=State!
   zip=ZIP Code
   thanks=Thank you for registering!
   subtext=[0] alphabetically comes before [1]
   ```

   The location of the locale directory is not important, but you must be able to add that location to the compiler's source path.

2. Create a locale/es_ES/RegistrationForm.properties file that contains the following text. Ensure that you save this file with UTF-8 encoding.

   ```
   registration_title=Registro
   submit=Someta La Forma
   personname=Nombre
   street_address=Dirección De la Calle
   city=Ciudad
   state=Estado
   zip=Código postal
   thanks=¡Gracias por colocarse!
   subtext=[0] viene alfabéticamente antes [1]
   ```

3. Create a LocalizedForm.mxml application that contains the following code:

```
<?xml version="1.0"?>
<!-- l10n/LocalizedForm.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.resources.ResourceBundle;
     import mx.controls.Alert;

     [ResourceBundle("RegistrationForm")]
     private static var rb:ResourceBundle;

     private function registrationComplete():void {
        Alert.show(rb.getString('thanks'));
     }

  ]]></mx:Script>

  <mx:Form>
     <mx:FormItem label="@Resource(key='personname',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='street_address',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='city',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='state',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='zip',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
  </mx:Form>
  <mx:Button id="b1" label="@Resource(key='submit',
bundle='RegistrationForm')" click="registrationComplete()"/>

</mx:Application>
```

4. Use the following mxmlc command line to compile the application with the en_US resource bundle:

```
mxmlc -locale en_US source-path+=path_to_locale_dir/{locale}
  -o Form-us.swf LocalizedForm.mxml
```

5. Use the following mxmlc command line to compile the application with the es_ES resource bundle:

```
mxmlc -locale es_US source-path+=path_to_locale_dir/{locale}
   -o Form-es.swf LocalizedForm.mxml
```

6. Run the two SWF files to see the different form labels and alerts in the applications.

Using the ResourceBundle API

The first step in localizing Flex code is to use the mx.resources.ResourceBundle API to replace all of the hard-coded content in the application that you want to localize. You can use the ResourceBundle API in ActionScript or MXML code.

All content that you localize must be removed from the main application. You implement calls to the content by accessing the resources bundles with the `[ResourceBundle]` metadata tag in ActionScript or the `@Resource` directive in MXML.

Using the ResourceBundle metadata tag

You must place the `[ResourceBundle]` metadata tag directly above a variable of type mx.resources.ResourceBundle, as the ActionScript code in the following example shows:

```
...
[ResourceBundle("bundlename")]
private static var rb:ResourceBundle;
...
```

The *`bundlename`* refers to a properties file in your compiler's source path that stores the localized data. You cannot create a ResourceBundle instance in ActionScript by using the new keyword; you must use the `[ResourceBundle]` metadata tag.

The compiler instantiates the variable. In your code, you can use the ResourceBundle API to access keys in the resource bundle. This API includes the `ResourceBundle.getString()` and `ResourceBundle.getObject()` methods. The following example uses the `[ResourceBundle]` metadata tag and uses the `getString()` method to access the contents of resource bundles:

```
<?xml version="1.0"?>
<!-- l10n/LocalizedFormGetString.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.resources.ResourceBundle;
     import mx.controls.Alert;

     [Bindable]
     [ResourceBundle("RegistrationForm")]
     private var rb:ResourceBundle;

     private function registrationComplete():void {
        Alert.show(rb.getString('thanks'));
     }

  ]]></mx:Script>

  <mx:Button id="b1" label="{rb.getString('submit')}"
click="registrationComplete()"/>

</mx:Application>
```

For information on accessing objects with the `getObject()` method, see "Using localized ActionScript classes" on page 981.

Using the @Resource directive

You can use the `@Resource` directive in your MXML tags instead of the `[ResourceBundle]` metadata tag. You specify a resource bundle name and a key in the `@Resource` directive to get a value from the resource bundle. If you specify only the key, the current class name is used as the bundle name. The *bundlename* is the name of the properties file or ResourceBundle subclass that contains the key. These are the two forms of the `@Resource` directive:

- `@Resource("key")`
- `@Resource(bundle="bundlename", key="key")`

The *bundlename* refers to a properties file in your compiler's source path that stores the localized data.

You use the `@Resource` directive to access a key in a resource bundle, as the following example shows:

```
<?xml version="1.0"?>
<!-- l10n/LocalizedFormResourceDirective.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Form>
     <mx:FormItem label="@Resource(key='personname',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='street_address',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='city', bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='state',
bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
     <mx:FormItem label="@Resource(key='zip', bundle='RegistrationForm')">
        <mx:TextInput/>
     </mx:FormItem>
  </mx:Form>
  <mx:Button id="b1" label="@Resource(key='submit',
bundle='RegistrationForm')"/>

</mx:Application>
```

Substituting localized String values

You can use String substitution in localized String values just as you would in any String. For example, the properties file could contain the following key:

```
subtext=[0] alphabetically comes before [1]
```

The following example uses the `StringUtil.substitute()` method to insert values into the localized String in ActionScript:

```
<?xml version="1.0"?>
<!-- l10n/StringSubstitution.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.resources.ResourceBundle;
     import mx.utils.StringUtil;

     [Bindable]
     [ResourceBundle("RegistrationForm")]
     private var rb:ResourceBundle;

     [Bindable]
     private var s:String;

     private function initApp():void {
        s = rb.getString('subtext');
        s = StringUtil.substitute(s, "Anant", "Nick");
     }
  ]]></mx:Script>

  <mx:Label id="l1" text="{s}"/>

</mx:Application>
```

Using localized ActionScript classes

You can use any ActionScript class as a localized resource. For example, you can have two different classes, one for each locale, that perform localized tasks in your Flex application. Rather than creating a properties file that acts as the repository of strings for the resource bundle, you create a class that references all the classes in the resource bundle. You extract a reference to the localized class from this new class, similar to the way you extract a String from a properties file.

To use localized classes, you must perform the following steps:

1. Embed the ResourceBundle in your Flex application by using the `[ResourceBundle]` metadata tag.
2. Use the `ResourceBundle.getObject()` method in your Flex application.
3. Create a bundle class that returns an Object that references all the localized class resources.
4. Store your localized ActionScript class in the locale/{locale} directories.

You can also use localized embedded assets, such as images and sound files, by including them in the resource bundles. To do this, you perform all the steps described in this section and the step described in "Using embedded assets" on page 983.

Using the getObject() method

To use localized objects or embedded assets, such as images, in an application, you get a reference to the resource bundle using the `[ResourceBundle]` metadata tag. You then call the `ResourceBundle.getObject()` method to get a particular class from the resource bundle, as the following example shows:

```
<?xml version="1.0"?>
<!-- l10n/SimpleClassApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">

  <mx:Script><![CDATA[
     import mx.resources.ResourceBundle;

     [Bindable]
     [ResourceBundle("MyObjectClassesBundle")]
     private var rb:ResourceBundle;

     private function initApp():void {
        var scClass:Class = Class(rb.getObject("SimpleClass"));
        var sc:SimpleClass = new scClass();

        b1.label = sc.returnString();
     }

  ]]></mx:Script>

  <mx:Button id="b1"/>

</mx:Application>
```

In this example, MyObjectClassesBundle is the name of a ResourceBundle subclass that you create to embed your localized ActionScript classes.

Creating a bundle class

To use embedded assets in a resource bundle, you must create a bundle class. This class must extend the ResourceBundle subclass. You typically include a constructor that only calls the `super()` method. This class also returns an object in the `getContent()` method. This object references all the classes that are required in the ResourceBundle.

You create one version of this class for each supported locale and store it in the locale/{locale} directories.

The following example shows the MyObjectClassesBundle class, which imports the localized ActionScript classes and adds a reference to them in the returned content object:

```
package {
import mx.resources.ResourceBundle;
import MyBird;
import SimpleClass;

public class MyObjectClassesBundle extends ResourceBundle {
  public function MyObjectClassesBundle() {
     super();
  }

  override protected function getContent():Object {
     var contentObj:Object = new Object();

     // Add a reference to the localized ActionScript class.
     contentObj["SimpleClass"] = SimpleClass;

     return contentObj;
  }
}
}
```

In this example, the localized ActionScript class is a simple class that has a method that returns a String, as the following example shows:

```
package {

  public class SimpleClass {

     // Constructor
     public function SimpleClass() {
     }

     public function returnString():String {
        return "hello from SimpleClass";
     }
  }

}
```

You can store the localized ActionScript class files in your locale/{locale} directories, or any directory that is accessible by the bundle class.

Using embedded assets

You can use embedded classes in resource bundles. These embedded assets can include images, sound files, or any type of file that can be embedded in a Flex application.

When you include embedded assets in resource bundles, you must create an ActionScript wrapper class for each embedded asset or object. You add the wrapper class for each localized embedded asset to your locale/{locale} directories.

The wrapper class extends a class that matches the kind of asset. For an image, the wrapper class typically extends BitmapAsset. For more information on other asset types and their associated classes that you can use in a resource bundle, see "Embedding asset types" on page 1121.

In your wrapper class, you must include only an empty constructor and an `[Embed]` metadata tag. In this class, you must use the class-based version of Embed and not the `@Embed` directive.

For example, to localize a GIF image, you create the following MyBird class:

```
package {
  import mx.core.BitmapAsset;
  [Embed(source='bird.gif')]
  public class MyBird extends BitmapAsset {
     public function MyBird() {
        // Empty constructor.
     }
  }
}
```

Your bundle class is the same as the one you created to use localized ActionScript classes (see "Creating a bundle class" on page 982).

```
package {
import mx.resources.ResourceBundle;
import MyBird;
import SimpleClass;

public class MyEmbeddedClassesBundle extends ResourceBundle {
  public function MyEmbeddedClassesBundle() {
     super();
  }

  override protected function getContent():Object {
     var contentObj:Object = new Object();

     // Add a reference to the embedded graphic asset.
     contentObj["MyBird"] = MyBird;

     // Add a reference to the localized ActionScript class.
     contentObj["SimpleClass"] = SimpleClass;

     return contentObj;
  }
}
}
```

In your Flex application, you instantiate the ResourceBundle the same way you did when using localized ActionScript classes. You also use the `getObject()` method to get the embedded asset. The following example gets an image class named MyBird:

```
<?xml version="1.0"?>
<!-- l10n/EmbeddedAssetApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.resources.ResourceBundle;

     [Bindable]
     [ResourceBundle("MyEmbeddedClassesBundle")]
     private var rb2:ResourceBundle;

     [Bindable]
     [ResourceBundle("MyObjectClassesBundle")]
     private var rb:ResourceBundle;

     [Bindable]
     private var bird:Object = rb2.getObject("MyBird");

     private function initApp():void {
        var sc:Object = rb.getObject("SimpleClass");
        b1.label = sc.returnString();
     }

  ]]></mx:Script>

  <mx:Panel id="panel1" title="Image from Resource Bundle">
     <mx:Image id="image1" source="{bird}"/>
  </mx:Panel>

  <mx:Button id="b1"/>

</mx:Application>
```

To make the conversion to run-time loading easier, you should not make direct references to localized classes. Because of this, you should add any visual children through ActionScript, as the following example shows:

```
<?xml version="1.0"?>
<!-- l10n/EmbeddedAssetAppAddChild.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.resources.ResourceBundle;
     import mx.controls.Image;

     [Bindable]
     [ResourceBundle("MyEmbeddedClassesBundle")]
     private var rb2:ResourceBundle;

     [Bindable]
     [ResourceBundle("MyObjectClassesBundle")]
     private var rb:ResourceBundle;

     [Bindable]
     private var bird:Object = rb2.getObject("MyBird");

     private function initApp():void {
        var sc:Object = rb.getObject("SimpleClass");
        b1.label = sc.returnString();

        var i:Image = new Image();
        i.source = bird;
        panel1.addChild(i);
     }

  ]]></mx:Script>

  <mx:Panel id="panel1" title="Image from Resource Bundle">
  </mx:Panel>

  <mx:Button id="b1"/>

</mx:Application>
```

Using localized properties files and ResourceBundle subclasses

You can use localized properties files and ResourceBundle subclasses in several ways.

Properties file formats and directories

You place string properties that you want to localize in properties files in a user-created locale directory. You can create one properties file per class, place consolidated keys and values in one file, or combine the two approaches.

To avoid specifying the bundle parameter of the `@Resource` MXML directive, you can give a properties file the same name as its corresponding class. However, it is good practice to specify the bundle parameter. If you don't specify the bundle parameter, the properties file location must be the same as the class, except that the root directory must be the name of the locale. For example, the properties file for /myApp/MyAlert.as is located in /fr_FR/myApp/MyAlert.properties, where fr_FR is a locale. Similarly, you could include the same properties file in multiple locale directories. You can set the locale by using the compc `locale` option. You cannot set the locale by using the Locale class at run time.

You include the locale directory in the compc ActionScript source path, but not in the ActionScript source path that the mxmlc command-line compiler uses when building the main SWF file.

Localization properties files follow the Java properties file format, which is a simple text format for storing keys and values. The following example shows the text strings for keys and values in the properties file:

```
registration_title=Registration
submit=Submit Form
personname=Name
street_address=Street Address
city=City
state=State!
zip=ZIP Code
thanks=Thank you for registering!
subtext=[0] alphabetically comes before [1]
```

All strings in properties files must be Latin-1 or UTF-8 encoded.

Localized properties file and class search order

You add localized properties files and ResourceBundle subclasses to directories that correspond to the locale name. For more information about localized ActionScript classes, see "Using localized ActionScript classes" on page 981.

When compiling SWC files, the mxmlc or compc compiler searches for files according to typical locale rules: the exact locale name, the locale name without a variant and without a country, and then en_EN.

The following table shows a properties file search order if the locale is set to fr_FR on the mxmlc or compc command line:

Localized file	Description
/en_EN/Grape.as	This file is not used because there is a version of the same file in the fr_FR directory, which is higher in the search order.
/en_EN/AppCommon.properties	This file is used because there is no other version of it higher up in the search order.
/fr/Grape.as	This file is not used because there is a version of it in the fr_FR directory, which is higher in the search order.
/fr/Apple.as	This file is used because there is no other version of it higher up in the search order.
/fr_FR/Grape.as	This file is used because it is at the top of the search order.
/fr_FR/Orange.as	This file is used because it is at the top of the search order.

Localizing Flex Software Development Kit resource bundles

In addition to your own code, you can localize Flex Software Development Kit (SDK) resource bundles the same way that you localize your own code. The Flex 2 SDK properties files for the en_EN locale are located in the Flex Framework 2/frameworks/locale/en_EN directory of your Flex Builder installation. These are the properties files that Adobe used to create the default Flex 2 SDK SWC file, framework_rb.swc. The en_EN directory includes a properties file named SharedResources.properties. This file contains keys and values for shared Flex framework strings and symbols, such as those used in dates, validators, and formatters. The framework properties are compiled into a framework_rb.swc file. You can create your own framework_rb.swc file by using the build_framework.xml Ant file.

Creating the application SWF file

You create an application SWF file with the mxmlc compiler. You must create one SWF file for each locale. The following example shows an mxmlc command line for creating a localized HelloWorld.swf file for the en_EN locale:

```
mxmlc -locale en_EN -source-path locale/{locale} HelloWorld.mxml
```

If you are using Flex Data Services to compile the application, you uncomment the <locale> option in the flex-config.xml file, and specify the locale, as the following example shows:

```
<!-- Specifies the locale for internationalization. -->
<locale>en_US</locale>
```

You can then use the locale token to add the location of the locale files in the `<source-path>`, as the following example shows:

```
<!-- list of path elements that form the roots of ActionScript class
  hierarchies -->
<source-path>
  <path-element>./user_classes</path-element>
  <path-element>./locale/{locale}</path-element>
</source-path>
```

Use the following mxmlc options to create a localized SWF file:

mxmlc option	Description
`library-path` *`path-element [...]`*	Links the SWC files to the resulting application SWF file. The compiler links in only those classes for the SWC file that are required. Specify this option with the `{locale}` token, which you use where the locale name is located. The default value of the `library-path` option includes all SWC files in the libs directory and the framework_rb.swc file. To point to individual classes or packages instead of entire SWC files, use the `source-path` option. If you set the value of the `library-path` as an option of the command line compiler, you must also explicitly add the framework.swc and framework_rb.swc files. Your new entry is not appended to the `library-path`, but replaces it. Use the += operator to append the new argument to the list of existing SWC files.
`locale` *`string`*	Specifies the locale that is packaged in the SWF file. You run the mxmlc compiler multiple times to create SWF files for more than one locale, and change only the `locale` and `output` options.
`output` *`string`*	Specifies the output path and filename for the resulting file. If you omit this option, the compiler saves the SWF file to the directory where the target file is located. The default SWF filename matches the target filename, but with a SWF file extension. If you use a relative path to define the *filename*, it is always relative to the current working directory, not the target MXML application root. The compiler creates extra directories based on the specified filename if those directories are not present.
`source-path` *`path-element [...]`*	Specifies the user-created localization directory. Specify this with the `{locale}` token, which you use where the locale name is located. For example, if you have files in /myfiles/locale_dir/en_EN and in /myfiles/locale_dir/ja_JP, specify a `source-path` value of /myfiles/locale_dir/{*`locale`*}.

For more information about the mxmlc compiler, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Creating a localized SWC file

You can package libraries of localized properties files and ActionScript classes in a SWC file that you create with the compc compiler. However, the SWC file is separate from your main application. It is not necessary to create a SWC file for the localized properties files if you are not building a library.

Use the following compc options to create a localized SWC file:

compc option	Description
`source-path`	Specifies the user-created localization directory. Specify this with the {locale} token, which you use where the locale name is located. For example, if you have files in /myfiles/locale_dir/en_EN and in /myfiles/locale_dir/ja_JP, specify an `source-path` value of /myfiles/locale_dir/{locale}.
`include-resource-bundles`	Specifies the resource bundles that you include in the localized SWC file. This list can have properties filenames (the name should not include the suffix or the base filepath) and or the names of ResourceBundle subclasses. When building a library with compc, you can get the list of resource bundles by using the `resource-bundle` option.
`library-path` *`path-element [...]`*	Links the SWC files to the resulting application SWF file. The compiler links in only those classes for the SWC file that are required. You can specify this with the {locale} token, which you use where the locale name is located. The default value of the `library-path` option includes all SWC files in the libs directory and the framework_rb.swc file. To point to individual classes or packages rather than entire SWC files, use the `source-path` option. If you set the value of the `library-path` as an option of the command-line compiler, you must also explicitly add the framework.swc and framework_rb.swc files. Your new entry is not appended to the `library-path`, but replaces it. You can use the += operator to append the new argument to the list of existing SWC files.
`output` *`filename`*	Specifies the fully qualified name of the SWC file. You always must specify this option to create a SWC file.

compc option	Description
`locale` *`string`*	Specifies the locale that is packaged in the SWC file. You run the compc compiler multiple times to create SWC files for more than one locale, and change only the `locale` and `output` options.
`resource-bundle-list` *`filename`*	Prints a list of resource bundles to input to the compc compiler to create a resource bundle SWC file. Do not use this option when creating the resource bundle SWC file. The *`filename`* argument is the name of the file that contains the list of bundles.

The following example shows a compc command line for creating a localized SWC file with the fr_FR resource bundle:

```
compc -locale fr_FR -source-path locale/{locale} -include-resource-bundles
  HelloWorld -output locale/fr_FR/HelloWorld.swc
```

For more information about the compc compiler, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

PART 4

Flex Programming Topics

4

This part introduces you to advanced programming techniques that you can use to make your applications more interactive and expressive.

The following topics are included:

CHAPTER 26

Dynamically Repeating Controls and Containers

26

This topic describes how to use Repeater components. Repeater components let you dynamically repeat MXML content at run time.

Flex also supports list controls, which provide better performance when you display large amounts of data. For more information about the HorizontalList, TileList, and List controls, see Chapter 12, "Using Data-Driven Controls," on page 439.

Contents

About Repeater components

Repeater components are useful for repeating a small set of simple user interface components, such as RadioButton controls and other controls typically used in Form containers. Repetition is generally controlled by an array of dynamic data, such as an Array object returned from a web service, but you can use static arrays to emulate simple `for` loops.

Although Repeater components look like containers in your code, they are not containers and have none of the automatic layout functionality of containers. Their sole purpose is to specify a series of subcomponents to include in your application one or more times based on the contents of a specified data provider. To align items that a repeater generates or perform any other layout task, place the Repeater component and its contents inside a container and apply the layout to that container.

Flex also supports HorizontalList, TileList, and List controls that provide better performance when you are displaying large amounts of data. Unlike the Repeater component, which instantiates all objects that are repeated, the HorizontalList, TileList, and List controls instantiate only objects visible in the list. If your data extends past a single screen or the visible space within any of its containers, you should use one of the list controls.

The HorizontalList control is a list control that displays data horizontally, much like the HBox container. The HorizontalList control always displays items from left to right. For more information, see "HorizontalList control" on page 450.

The TileList control is a list control that displays data in a tile layout, much like the Tile container. The TileList control provides a `direction` property that determines if the next item is down or to the right. For more information, see "TileList control" on page 454.

The List control displays data in a single vertical column. For more information, see "List control" on page 440.

Using the Repeater component

You use the `<mx:Repeater>` tag to declare a Repeater component that handles repetition of one or more user interface components based on dynamic or static data arrays at run time. The repeated components can be controls or containers. Using a Repeater component requires data binding to allow for run-time-specific values. For more information about data binding, see Chapter 39, "Storing Data," on page 1269.

You can use the `<mx:Repeater>` tag anywhere a control or container tag is allowed. To repeat user interface components, you place their tags within the `<mx:Repeater>` tag. All components derived from the UIComponent class can be repeated with the exception of the `<mx:Application>` container tag. You can also use more than one `<mx:Repeater>` tag in an MXML document, and you can nest `<mx:Repeater>` tags.

Declaring the Repeater component in MXML

You declare the Repeater component in the `<mx:Repeater>` tag. The following table describes the Repeater component's properties:

The following table describes the Repeater component's events:

Property	Description
`id`	Instance name of the corresponding Repeater component.
`dataProvider`	An implementation of the ICollectionView interface, IList interface, or Array class, such as an ArrayCollection object. You must specify a `dataProvider` value or the Repeater component will not execute. Generally, you specify the value of the `dataProvider` property as a binding expression because the value is not known until run time.
`startingIndex`	Number that specifies the element in the data provider at which the repetition starts. The data provider array is zero-based, so to start at the second element of the array, specify a starting index of one. If the `startingIndex` is not within the range of the `dataProvider` property, no repetition occurs.
`count`	Number that specifies how many repetitions occur. If the `dataProvider` property has fewer items than the number in the `count` property, the repetition stops with the last item.
`currentIndex`	Number that specifies the element of the `dataProvider` item currently being processed. The data provider array is zero-based, so when the third element is being processed, the current index is two. This property changes as the Repeater component executes, and is -1 after the execution is complete. It is a read-only property that you cannot set in the `<mx:Repeater>` tag.
`currentItem`	Reference to the item that is being processed in the `dataProvider` property. This property changes as the Repeater component executes, and is null after the execution is complete. It is a read-only property that you cannot set in the `<mx:Repeater>` tag. After a Repeater component finishes repeating, you do not use the `currentItem` property to get the current item. Instead, you call the `getRepeaterItem()` method of the repeated component itself. For more information, see "Event handlers in Repeater components" on page 1007.
`recycleChildren`	Boolean value that, when set to `true`, binds new data items into existing Repeater children, incrementally creates new children if there are more data items, and destroys extra children that are no longer required. For more information, see "Recreating children in a Repeater component" on page 1016.

Event	Description
`repeat`	Dispatched each time an item is processed and `currentIndex` and `currentItem` are updated.
`repeatEnd`	Dispatched after all the subcomponents of a repeater are created.
`repeatStart`	Dispatched when Flex begins processing the `dataProvider` property and begins creating the specified subcomponents.

Basic principles of the Repeater component

The simplest repeating structures you can create with a Repeater component are static loops that execute a set number of times. The following Repeater component emulates a simple for loop that executes four times, printing a simple line of text and incrementing the counter by one each time:

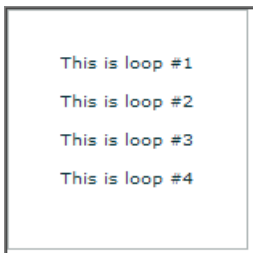
```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*">

  <mx:Script>
    <![CDATA[
      [Bindable]
      public var myArray:Array=[1,2,3,4];
    ]]>
  </mx:Script>

  <mx:ArrayCollection id="myAC" source="{myArray}"/>
  <mx:Repeater id="myrep" dataProvider="{myAC}">
    <mx:Label id="Label1" text="This is loop #{myrep.currentItem}"/>
  </mx:Repeater>
</mx:Application>
```

In actuality, the counter is not the data within the array, but the position within the array. As long as the array contains four elements, you can provide any data within the array itself, and the Repeater component still executes four times. The following example illustrates this principle:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            [Bindable]
            public var myArray:Array=[10,20,30,40];
        ]]>
    </mx:Script>

    <mx:ArrayCollection id="myAC" source="{myArray}"/>

    <mx:Repeater id="myrep" dataProvider="{myAC}">
        <mx:Label id="Label1" text="This is loop #{myrep.currentIndex+1}"/>
  </mx:Repeater>
</mx:Application>
```

Notice that this example prints the current index plus one, not the index itself. This is because the first element of the array is assigned an index value of zero (0).

Both of the previous examples yield the following identical result:

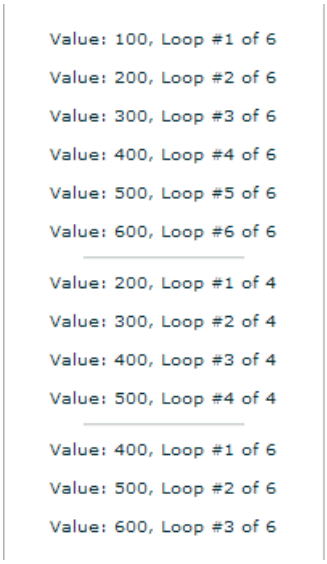


You can also use the `startingIndex` and `count` properties to adjust the starting point and total number of executions. The `count` property sets a maximum on the number of times the Repeater component executes. The `count` property is often—but not always—an exact measure of the number of times that the content within the Repeater component will execute. This number is because the Repeater component stops after the last element of the data provider is reached, regardless of the value of the `count` property.

The following example shows how the `count` and `startingIndex` properties affect a Repeater component:

```
<?xml version="1.0"?>
<!-- repeater\StartingIndexCount.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">


    <mx:Script>
        <![CDATA[
            [Bindable]
            public var myArray:Array=[100,200,300,400,500,600];
        ]]>
    </mx:Script>

    <mx:ArrayCollection id="myAC" source="{myArray}"/>

    <mx:Repeater id="myrep" dataProvider="{myAC}" count="6">
        <mx:Label id="Label1"
            text="Value: {myrep.currentItem}, Loop #{myrep.currentIndex+1}
                of {myrep.count}"/>
    </mx:Repeater>

    <mx:HRule/>

    <mx:Repeater id="myrep2" dataProvider="{myAC}"
            count="4" startingIndex="1">
        <mx:Label id="Label2"
            text="Value: {myrep2.currentItem},
                Loop #{myrep2.currentIndex-myrep2.startingIndex+1}
                of {myrep2.count}"/>
    </mx:Repeater>

    <mx:HRule/>

    <mx:Repeater id="myrep3" dataProvider="{myAC}" count="6"
            startingIndex="3">
        <mx:Label id="Label3"
            text="Value: {myrep3.currentItem},
                Loop #{myrep3.currentIndex-myrep3.startingIndex+1}
                of {myrep3.count}"/>
     </mx:Repeater>
</mx:Application>
```

This example produces the following output:



The first Repeater component loops through each element of the data provider, starting with the first and stopping after the last. The second Repeater component starts at the second element of the data provider and iterates four times, ending at the fifth element. The third Repeater component starts with the fourth element and continues until the end of the data provider array, and then stops. Only three iterations occur despite the fact that the `count` property is set to 6.

Creating dynamic loops with the Repeater component

The principles described in "Basic principles of the Repeater component" on page 998 work with dynamic data providers as well. The only difference is the source of the data array; instead of a static array written directly into your application, the array is defined in an `<mx:Model>` tag and drawn from an XML file, a web service, a remote object, or some other source. The data is collected and evaluated at run time to determine the number and value of elements in the data provider and how the Repeater component behaves.

In the following example, an <mx:Repeater> tag repeats a RadioButton control for each product in an XML file:

```
<?xml version="1.0"?>
<!-- repeater\DynamicLoop.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="catalogService.send()">
    <mx:HTTPService id="catalogService" url="assets/catalog.xml"
            resultFormat="e4x"/>
    <mx:XMLListCollection id="myXC"
            source="{catalogService.lastResult.product}"/>
    <mx:Repeater id="r" dataProvider="{myXC}">
        <mx:RadioButton id="Radio" label="{r.currentItem.name}"
                width="150"/>
    </mx:Repeater>
</mx:Application>
```

Assume that catalog.xml contains the following:

```
<?xml version="1.0"?>
<!-- repeater\assest\catalog.xml -->
<products>
  <product>
    <name>Name</name>
    <price>Price</price>
    <freeship>Free Shipping?</freeship>
  </product>
  <product>
    <name>Whirlygig</name>
    <price>5</price>
    <freeship>false</freeship>
  </product>
  <product>
    <name>Tilty Thingy</name>
    <price>15</price>
    <freeship>true</freeship>
  </product>
<product>
    <name>Really Big Blocks</name>
    <price>25</price>
    <freeship>true</freeship>
  </product>
</products>
```

Flex displays the following output:

You can still use the `count` property to restrict the number of iterations performed. You can use the `startingIndex` property to skip entries at the beginning of the data provider.

In the preceding example, the first product entry in the XML file contains metadata that other applications use to create column headers. Because it doesn't make sense to include that entry in the radio buttons, start the Repeater component at the second element of the data provider, as in the following code example:

```
<?xml version="1.0"?>
<!-- repeater\StartSecondElement.mxml -->
<mx:Application     xmlns:mx="http://www.adobe.com/2006/mxml">
<mx:HTTPService id="catalogService" url="assets/catalog.xml"
            resultFormat="e4x"/>
    <mx:XMLListCollection id="myXC"
            source="{catalogService.lastResult.product}"/>
    <mx:Repeater id="r" dataProvider="{myXC}"
            startingIndex="1">
        <mx:RadioButton id="Radio" label="{r.currentItem.name}"
            width="150"/>
    </mx:Repeater>
</mx:Application>
```

The preceding example produces the following output:

Referencing repeated components

To reference individual instances of a repeated component, you use indexed `id` references, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
<!-- repeater\RefRepeatedComponents.mxml -->
    <mx:Script>
        <![CDATA[
            public function labelTrace():void {
                for (var i:int = 0; i < nameLabel.length; i++)
                    trace(nameLabel[i].text);
            }
        ]]>
    </mx:Script>

    <mx:Model id="catalog" source="assets/catalog.xml"/>
    <mx:ArrayCollection id="myAC" source="{catalog.products.product}"/>
    <mx:Label id="title" text="Products:"/>
    <mx:Repeater id="r" dataProvider="{myAC}" startingIndex="1">
        <mx:Label id="nameLabel"
            text="{r.currentItem.name}: ${r.currentItem.price}"
            width="200"/>
    </mx:Repeater>
    <mx:Button label="Trace" click="labelTrace();"/>
</mx:Application>
```

In this example, the `id` of the repeated Label control is `nameLabel`; each `nameLabel` instance created has this `id`. You reference the individual Label instances as `nameLabel[0]`, `nameLabel[1]`, and so on. You reference the total number of `nameLabel` instances as `nameLabel.length`. The `for` loop traces the `text` property of each Label control in the `nameLabel` Array object. The `labelTrace()` method prints the name and price of each product in the system log, provided you've defined it in the `mm.cfg` file.

Referencing repeated child components

When a container is repeated and indexed in an array, its children are also indexed. For example, for the following MXML code, you reference the child Label controls of the VBox container `vb[0]` as `nameLabel[0]` and `shipLabel[0]`. The syntax for referencing the children is the same as the syntax for referencing the parent.

```
<?xml version="1.0"?>
<!-- repeater\RefRepeatedChildComponents.mxml -->
<mx:Application borderStyle="solid" width="300" height="300"
xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function labelTrace():void {
                for (var i:int = 0; i < nameLabel.length; i++)
                    trace(nameLabel[i].text);
            }
        ]]>
    </mx:Script>

    <mx:Model id="catalog" source="assets/catalog.xml"/>
    <mx:Label id="title" text="Products:"/>

    <mx:Repeater id="r" dataProvider="{catalog.products.product}"
            startingIndex="1">
        <mx:VBox id="vb">
            <mx:Label id="nameLabel"
                text="{r.currentItem.name}: ${r.currentItem.price}"
                width="200"/>
            <mx:Label id="shipLabel"
                text="Free shipping: {r.currentItem.freeship}"/>
            <mx:Spacer/>
        </mx:VBox>
    </mx:Repeater>

    <mx:Button label="Trace" click="labelTrace();"/>
</mx:Application>
```

Referencing nested Repeater components

When `<mx:Repeater>` tags are nested, the inner `<mx:Repeater>` tags are indexed Repeater components. For example, for the following MXML code, you access the nested Repeater components as `r2[0]`, `r2[1]`, and so on. The syntax for referencing the children is same as the syntax for referencing the parent.

```
<?xml version="1.0"?>
<!-- repeater\RefNestedComponents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      public function labelTrace():void {
        for (var i:int = 0; i < nameLabel.length; i++)
         for (var j:int = 0; j < nameLabel[i].length; j++)
          trace(nameLabel[i][j].text);
      }
    ]]>
  </mx:Script>

  <mx:Model id="data">
    <color>
      <colorName>Red</colorName>
      <colorName>Yellow</colorName>
      <colorName>Blue</colorName>
    </color>
  </mx:Model>

  <mx:Model id="catalog" source="assets/catalog.xml"/>
  <mx:ArrayCollection id="myAC1" source="{catalog.products.product}"/>
  <mx:ArrayCollection id="myAC2" source="{data.colorName}"/>

  <mx:Label id="title" text="Products:"/>

  <mx:Repeater id="r" dataProvider="{myAC1}" startingIndex="1">
    <mx:Repeater id="r2" dataProvider="{myAC2}">
      <mx:Label id="nameLabel" text="{r2.currentItem}
          {r.currentItem.name}: ${r.currentItem.price}" width="200"/>
    </mx:Repeater>
  </mx:Repeater>

  <mx:Button label="Trace" click="labelTrace();"/>
</mx:Application>
```

This application places the full list of products with color and price into the system log when you click the Button control (provided you have defined the log file in the mm.cfg file) and yields the following on screen:

In the previous example, the instances of the Label control are multiply indexed because they are inside multiple Repeater components. For example, the index `nameLabel[1][2]` contains a reference to the Label control produced by the second iteration of `r` and the third iteration of `r2`.

Event handlers in Repeater components

When a Repeater component is busy repeating, each repeated object that it creates can bind at that moment to the Repeater component's `currentItem` property, which is changing as the Repeater component repeats. You cannot give each instance its own event handler by writing something like `click="doSomething({r.currentItem})"` because binding expressions are not allowed in event handlers, and all instances of the repeated component must share the same event handler.

Repeated components and repeated Repeater components have a `getRepeaterItem()` method that returns the item in the `dataProvider` property that was used to produce the object. When the Repeater component finishes repeating, you can use the `getRepeaterItem()` method to determine what the event handler should do based on the `currentItem` property. To do so, you pass the `event.currentTarget.getRepeaterItem()` method to the event handler. The `getRepeaterItem()` method takes an optional index that specifies which Repeater components you want when nested Repeater components are present; the 0 index is the outermost Repeater component. If you do not specify the index argument, the innermost Repeater component is implied.

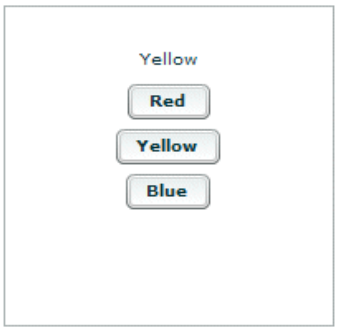
NOTE

After a Repeater component finishes repeating, you do not use the `Repeater.currentItem` property to get the current item. Instead, you call the `getRepeaterItem()` method of the repeated component itself.

The following example illustrates the `getRepeaterItem()` method. When the user clicks each repeated Button control, the corresponding `colorName` value from the data model appears in the Button control label.

```
<?xml version="1.0"?>
<!-- repeater\GetItem.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      public function clicker(cName:String):void {
         foolabel.text=cName;
       }
    ]]>
  </mx:Script>

  <mx:Label id="foolabel" text="foo"></mx:Label>

  <mx:Model id="data">
    <color>
      <colorName>Red</colorName>
      <colorName>Yellow</colorName>
      <colorName>Blue</colorName>
    </color>
  </mx:Model>
  <mx:ArrayCollection id="myAC" source="{data.colorName}"/>

  <mx:Repeater id="myrep" dataProvider="{myAC}">
    <mx:Button click="clicker(event.currentTarget.getRepeaterItem());"
      label="{myrep.currentItem}"/>
  </mx:Repeater>
</mx:Application>
```

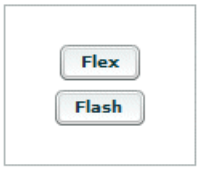
After the user clicks the Yellow button, the application looks like this:



The code in the following example uses the `getRepeaterItem()` method to display a specific URL for each Button control that the user clicks. The Button controls must share a common data-driven click handler, because you cannot use binding expressions inside event handlers. However, the `getRepeaterItem()` method lets you change the functionality for each Button control.

```
<?xml version="1.0"?>
<!-- repeater\DisplayURL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      [Bindable]
      public var dp:Array = [ { label: "Flex",
        url: "http://www.adobe.com/flex" },
        { label: "Flash", url: "http://www.adobe.com/flash" } ];
    ]]>
  </mx:Script>

  <mx:ArrayCollection id="myAC" source="{dp}"/>
  <mx:Repeater id="r" dataProvider="{myAC}">
    <mx:Button label="{r.currentItem.label}" click="navigateToURL(new
      URLRequest(event.currentTarget.getRepeaterItem().url));"/>
  </mx:Repeater>
</mx:Application>
```

When executed, this example yields two buttons: one for Flex and one for Flash. When you click the button of your choice, the relevant product page loads in a new browser window.



Accessing specific instances of repeated components

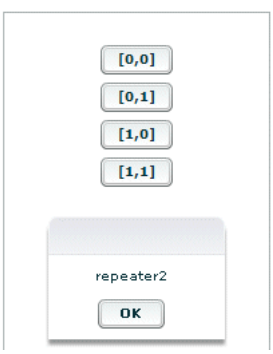
Repeated components and repeated Repeater components have three properties that you can use to dynamically keep track of specific instances of repeated objects, to determine which Repeater component produced them, and to determine which `dataProvider` items were used by each Repeater component. The following table describes these properties:

Property	Description
`instanceIndices`	Array that contains the indices required to reference the component from its document. This Array is empty unless the component is in one or more Repeater components. The first element corresponds to the outermost Repeater component. For example, if the `id` is `b` and `instanceIndices` is `[2,4]`, you would reference it on the document as `b[2][4]`.
`repeaters`	Array that contains references to the Repeater components that produced the component. The Array is empty unless the component is in one or more Repeater component. The first element corresponds to the outermost Repeater component.
`repeaterIndices`	Array that contains the indices of the items in the `dataProvider` properties of the Repeater components that produced the component. The Array is empty unless the component is within one or more Repeater components. The first element corresponds to the outermost Repeater component. For example, if `repeaterIndices` is `[2,4]`, the outer Repeater component used its `dataProvider[2]` data item and the inner Repeater component used its `dataProvider[4]` data item. This property differs from `instanceIndices` if the `startingIndex` of any of the Repeater components is not 0. For example, even if a Repeater component starts at `dataProvider` item 4, the document reference of the first repeated component is `b[0]`, not `b[4]`.

The following example application uses the `repeaters` property to display the `id` value of the Repeater components in an Alert control when the user clicks one of the Button controls labelled by the index value of the outer Repeater component and inner Repeater component, respectively. It uses the `repeaters` property to get the `id` value of the inner Repeater component:

```
<?xml version="1.0"?>
<!-- repeater\RepeaterProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      import mx.controls.Alert;
      [Bindable]
      public var myArray:Array=[1,2];
    ]]>
  </mx:Script>

  <mx:ArrayCollection id="myAC" source="{myArray}"/>
  <mx:Repeater  id="repeater1" dataProvider="{myAC}">
    <mx:Repeater  id="repeater2" dataProvider="{myAC}">
      <mx:Button
        label="[{repeater1.currentIndex},{repeater2.currentIndex}]"
        click="Alert.show(event.target.repeaters[1].id);"/>
    </mx:Repeater>
  </mx:Repeater>
</mx:Application>
```

The previous example produces the following results when the user clicks the third button:

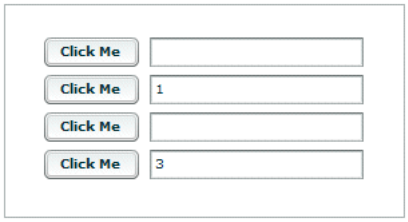


The following example application uses the `instanceIndices` property to set the `text` property of a TextInput control when the user clicks the corresponding Button control in the set of repeated Button and TextInput controls. You need to use the `instanceIndices` property because you must get the correct object dynamically; you cannot get it by its `id` value.

The following example shows how to use the `instanceIndices` property to set the `text` property of a TextInput control when the user clicks a Button control. The argument `event.target.instanceIndices` gets the index of the corresponding TextInput control.

```
<?xml version="1.0"?>
<!-- repeater\InstanceIndicesProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      [Bindable]
      public var myArray:Array=[1,2,3,4,5,6,7,8];
    ]]>
  </mx:Script>

  <mx:ArrayCollection id="myAC" source="{myArray}"/>
  <mx:Repeater id="list" dataProvider="{myAC}" count="4" startingIndex="2">
    <mx:HBox>
      <mx:Button label="Click Me"
        click="myText[event.target.instanceIndices].text=
        event.target.instanceIndices.toString();"/>
      <mx:TextInput id="myText"/>
    </mx:HBox>
  </mx:Repeater>
</mx:Application>
```

This example yields the following results when the user clicks the second and fourth buttons:



The following code shows how to use the `repeaterIndices` property instead of the `instanceIndices` property to set the `text` property of a TextInput control when the user clicks a Button control. The value of `event.target.repeaterIndices` is based on the current index of the Repeater component. Because the `startingIndex` property of the Repeater component is set to 2, it does not match the `event.target.instanceIndices` value, which always starts at 0.

```
<?xml version="1.0"?>
<!-- repeater\RepeaterIndicesProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      [Bindable]
      public var myArray:Array = [1,2,3,4,5,6,7,8];
    ]]>
  </mx:Script>

  <mx:Repeater id="list" dataProvider="{myArray}"
    startingIndex="2" count="4">
  <mx:HBox>
    <mx:Button id="b" label="Click Me"
      click="myText[event.target.repeaterIndices-list.startingIndex].text=
        event.target.repeaterIndices.toString();"/>
      <mx:TextInput id="myText"/>
    </mx:HBox>
  </mx:Repeater>
</mx:Application>
```

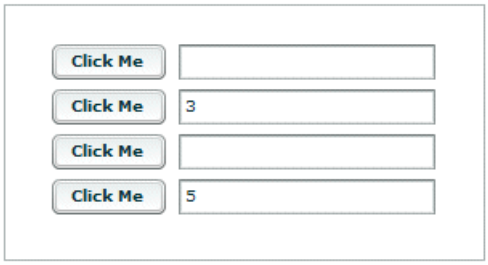
In this case, clicking the button prints the current element of the data provider, not the current iteration of the loop.

NOTE

The value of the `repeaterIndices` property is equal to the `startingIndex` property on the first iteration of the Repeater component. Subtracting the `startingIndex` value from the `repeaterIndices` value always yields the `instanceIndices` value. An alternative, equivalent `click` event would be:

```
click="myText[event.target.instanceIndices].text=event.target.repeaterIndices.to
   String()"/>
```

Either version yields the following results when the user clicks the second and fourth buttons:



Using a Repeater component in a custom MXML component

You can use the `<mx:Repeater>` tag in an MXML component definition in the same way that you use it in an application file. When you use the MXML component as a tag in another MXML file, the repeated items appear. You can access an individual repeated item by its array index number, just as you do for a repeated item defined in the application file.

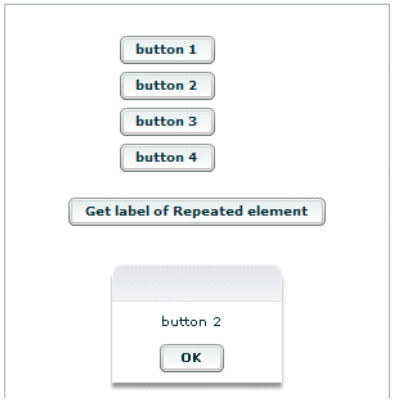
In the following example, a Button control in an MXML component called `childComp` is repeated for every element in an Array object called dp:

```
<?xml version="1.0"?>
<!-- repeater\myComponents\CustButton.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
    <![CDATA[
      [Bindable]
      public var dp:Array=[1,2,3,4];
    ]]>
  </mx:Script>

  <mx:ArrayCollection id="myAC" source="{dp}"/>

  <mx:Repeater id="r" dataProvider="{myAC}">
    <mx:Button id="repbutton" label="button {r.currentItem}"/>
  </mx:Repeater>
</mx:VBox>
```

The application file in the following example uses the childComp component to display four Button controls, one for each element in the array. The `getLabelRep()` function displays the label text of the second Button in the array.

```
<?xml version="1.0"?>
<!-- repeater\RepeatCustButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:MyComp="myComponents.*">

  <mx:Script>
    <![CDATA[
      import mx.controls.Alert;
      public function getLabelRep():void {
        Alert.show(comp.repbutton[1].label);
      }
    ]]>
  </mx:Script>

  <MyComp:CustButton id="comp"/>

  <mx:Button label="Get label of Repeated element" width="200"
    click="getLabelRep();"/>
</mx:Application>
```

The previous example produces the following output when the user clicks the last button:



Dynamically creating components based on data type

You can use a Repeater component to dynamically create different types of components for specific items in a set of data. A Repeater component broadcasts a `repeat` event as it executes, and this event is broadcast after the `currentIndex` and `currentItem` properties are set. You can call an event handler function on the `repeat` event, and dynamically create different types of components based on the individual data items.

How a Repeater component executes

A Repeater component executes initially when it is instantiated. If the Repeater component's `dataProvider` property exists, it proceeds to instantiate its children, and they instantiate their children, recursively.

The Repeater component re-executes whenever its `dataProvider`, `startingIndex`, or `count` properties are set or modified either explicitly in ActionScript, or implicitly by data binding. If the `dataProvider` property is bound to a web service result, the Repeater component re-executes when the web service operation returns the result. A Repeater component also re-executes in response to paging through the `dataProvider` property by incrementally increasing the `startingIndex` value, as the following example shows:

```
r.startingIndex += r.count;
```

When a Repeater component re-executes, it destroys any children that it previously created (assuming the `recycleChildren` property is set to `false`), and then reinstantiates its children based on the current `dataProvider` property. The number of children in the container might change, and the container layout changes to accommodate any changes to the number of children.

Recreating children in a Repeater component

The `recycleChildren` property controls whether children of a Repeater component are recreated when the Repeater component re-executes. When you set the `recycleChildren` property to `false`, the Repeater component recreates all the objects when you swap one `dataProvider` with another, or sort, which causes a performance lag. Only set this property to `true` if you are confident that modifying your `dataProvider` will not recreate the Repeater component's children.

The default value of the `recycleChildren` property is `false`, to ensure that you do not leave stale state information in a repeated instance. For example, suppose you use a Repeater component to display photo images, and each Image control has an associated NumericStepper control for how many prints you want to order. Some of the state information, the image, comes from the `dataProvider`, while other state information, the print count, is set by user interaction. If you set the `recycleChildren` property to `true` and page through the photos by incrementally increasing the Repeater component's `startingIndex` value, the Image controls bind to the new images, but the NumericStepper controls keep the old information.

Considerations when using a Repeater component

Consider the following when you use a Repeater component:

- You cannot use a Repeater component to iterate through a two-dimensional Array object that is programmatically generated. This is because the elements of an Array object do not trigger `changeEvent` events, and therefore cannot function as binding sources at run time. Binding copies initial values during instantiation after variables are declared in an `<mx:Script>` tag, but before `initialize` handlers are executed.
- Run-time changes to an array used as a data provider are not reflected in the Repeater component. Use a collection if you need to allow run-time modification.
- Forgetting curly braces ({ }) in a `dataProvider` property is a common mistake when using a Repeater component. If the Repeater component doesn't execute, make sure that the binding is correct.
- If repeated objects are displayed out of order and you are using adjacent or nested Repeater components, you might need to place a dummy UIComponent immediately after the Repeater that is displaying objects incorrectly.

  The code in the following example contains adjacent `<mx:Repeater>` tags and uses an `<mx:Spacer>` tag to create a dummy UIComponent:

```
<mx:VBox>
  <mx:Repeater id="r1">
    ...
  </mx:Repeater>
  <mx:Repeater id="r2">
    ...
  </mx:Repeater>
  <mx:Spacer height="0" id="dummy"/>
</mx:VBox>
```

The code in the following example contains nested `<mx:Repeater>` tags and uses an `<mx:Spacer>` tag to create a dummy UIComponent:

```
<mx:VBox>
  <mx:Repeater id="outer">
    <mx:Repeater id="inner">
      ...
    </mx:Repeater>
    <mx:Spacer id="dummy" height="0">
  </mx:Repeater>
</mx:VBox>
```

CHAPTER 27

Using View States

27

View states let you vary the content and appearance of a component, typically in response to a user action. To use view states, you define the base view state of a component, and one or more additional view states that specify modifications to the base view state.

This topic describes how to control component or application display by using view states. It does not describe how to apply transitions, which control how the application behaves while it changes from one view state to another. For information on transitions, see Chapter 28, "Using Transitions," on page 1051.

Contents

About view states

In many rich Internet applications, the interface changes based on the task the user is performing. A simple example is an image that changes when the user rolls the mouse over it. More complex examples include user interfaces whose contents change depending on the user's progress through a task, such as changing from a browse view to a detail view. These interfaces can use a smooth open-close effect to transition between views. View states let you easily implement such behaviors without requiring you to load multiple Macromedia Flash applications. The view states can simplify what would otherwise be complex event handling code.

View states also let you coherently structure your application to present a varying appearance and behavior by defining a base application or component (the base state) and sets of changes that modify the base, or alternatively, modify another state. Each set of changes defines a view state, which can add or remove children, set or change styles and properties, or define state-specific event listeners.

You can also define *transitions* between view states, which control how the application changes visually as it changes (transitions) from one view state to another. For information on how to use transitions, see Chapter 28, "Using Transitions," on page 1051.

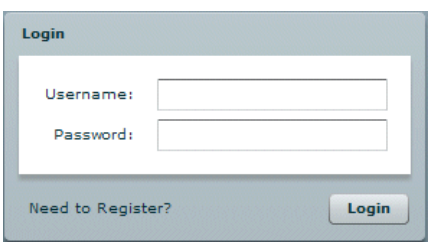
At its simplest, a view state is a particular view of a component. For example, a product thumbnail could have two view states; a base state with minimal information, and a "rich" state with additional information. Similarly, an application could have multiple view states that correspond to different application conditions, such as a login state, an overview state, or a search results state.

You do not have to define view states to create an interactive rich Internet application; however, they provide a powerful tool for logically structuring a dynamic application.

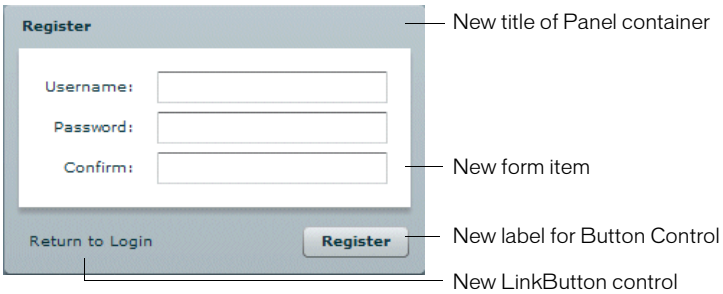
The following examples show ways of using view states.

Use case: Login interface

You can use view states to easily implement a login and registration form. In this case, the initial view state prompts the user to log in, and includes a link that lets them register, if necessary, as the following image shows:



If the user selects the Need to Register link, the form changes view state to display registration information, as the following image shows:



When the user clicks the Return to Login link, the view state changes back to the Login form. You can see the code for this example in "Example: Login form application" on page 1026.

Use case: Custom item renderer

A shopping application that displays multiple items on a page might have a custom thumbnail item renderer with two view states. In the base view state, the item cell might look the following image:

When the user rolls the mouse over the item, the view state changes: the thumbnail no longer has the availability and rating information, and now has buttons that let the user get more information or add the item to the wish list or cart. In the new state, the cell also has a border and a drop shadow, as the following image shows:

In this example, the application item renderer's two view states have different child components and have different component styles. The summary state, for example, includes an availability label and a star rating image, and has no border. The rolled-over state replaces the label and rating components with three buttons, and has an outset border.

You can see the code for an example similar to this in "Example: Using view states with a custom item renderer" on page 1043. For information on item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 851.

Defining and applying view states

This section describes the basic concepts for creating and applying view states, and concludes with a simple example that shows how to use these rules. Later sections provide more details on using view state classes, methods, and properties and on creating view state-based applications.

Applying view states

You can set the view state of any Flex component, including the Application component. A component can have different view states at different times, but is in only one view state at a time.

- The component's `currentState` property specifies its view state. As a general rule, you should use this property to change view states.
- You can also change a component's view state by calling the `setCurrentState()` method of the UIComponent class. Use this method when you do *not* want to apply a transition that you have defined between two view states.
- To specify the base view state, set the `currentState` property to `""`.
- Many applications change view states in response to events. Applications can also change the view state based on conditions such as the user status, including whether or not the user is logged in.

The following example code lets the user change between two view states by clicking Button controls; button b1 sets the view state to the newButton state, and button b2 sets the view state to the base state.

```
<mx:Button id="b1" label="Add a Button" click="currentState='newButton';"/>
<mx:Button id="b2" label="Remove Added Button" click="currentState='';"/>
```

Defining view states

The properties, styles, event listeners, and child components that you define for a component specify the component's base, or default, view state. You can define additional view states by using `<mx:State>` tags or State class objects. Each view state specifies changes to the base view state, or to another view state.

You define view states as follows:

- You can only define view states at the root of an application or of a custom component; that is, as a property of the `<mx:Application>` tag of an application file, or of the root tag of an MXML component.
- You define states by using the component's `states` property, normally as an `<mx:states>` child tag.
- You populate the `states` property with an Array of one or more States classes, where each State class instance corresponds to a view state.
- You use the `name` property of the State class to specify an identifier for the view state. To change to that view state, you set a component's `currentState` property to the value of the `name` property.
- To define the view state relative to another view state, specify the state in the `<mx:State>` `basedOn` property; otherwise the view state consists of overrides to the base view state. (When Flex changes states, it restores the base state, applies any changes from the state determined by the State class `basedOn` property, and then applies the changes defined in the new state.)
- In each State class, the `overrides` property contains an array of state overrides. The `overrides` property is the default States class property, so you can omit it, and the `<mx:Array>` tag, in MXML.
- You define each state override by using an element described in "Elements of a view state" on page 1025.

The following MXML code shows this structure:

```
<mx:Application> <!-- Can also be the root tag of a custom component. -->
  <mx:states>
    <mx:State>
      <mx:AddChild/>
      .
      <mx:SetStyle/>
      .
      <mx:SetProperty/>
      .
      <mx:SetEventHandler/>
      .
    </mx:State>
```

```
    <mx:State>
      .
      .
    </mx:State>
    .
    .
  </mx:states>
  .
  .
</mx:Application>
```

The following example shows a simple `<mx:states>` tag with a single State object:

```
<?xml version="1.0"?>
<!-- states\StatesSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewButton">
            <mx:AddChild relativeTo="{v1}">
                <mx:Button id="b2" label="New Button"/>
            </mx:AddChild>
            <mx:SetProperty target="{b1}" name="enabled" value="false"/>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1">
      <mx:Button label="Change State"
       click="currentState = currentState=='NewButton' ? '':'NewButton';"/>

      <mx:Button id="b1"/>
    </mx:VBox>
</mx:Application>
```

A single State object can modify multiple components by specifying the components to modify in the individual child tags' `target` properties. In the preceding example, the `<mx:AddChild>` tag adds a child to the v1 VBox control, and the `<mx:SetProperty>` tag changes a property of the b1 Button control. The following code changes the `enabled` property for two Button controls, setting one `false`, and the other `true`:

```
<?xml version="1.0"?>
<!-- states\StatesSimple2Buttons.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewButton">
            <mx:AddChild relativeTo="{v1}">
                <mx:Button id="b3" label="New Button"/>
            </mx:AddChild>
            <mx:SetProperty target="{b1}" name="enabled" value="false"/>
            <mx:SetProperty target="{b2}" name="enabled" value="true"/>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1">
      <mx:Button label="Change State"
       click="currentState = currentState=='NewButton' ? '':'NewButton';"/>

      <mx:Button id="b1"/>
      <mx:Button id="b2" enabled="false"/>
    </mx:VBox>
</mx:Application>
```

Elements of a view state

A view state is a named set of changes, or *overrides*, to a target component. By default, Flex applies the override to the application or to the custom component that defines the view state. You can use the following classes to define the overrides:

- Adding or removing child objects, by using the AddChild and RemoveChild classes; for example, the following lines add a Button child control to the v1 VBox control:

  ```
  <mx:AddChild relativeTo="{v1}">
    <mx:Button label="New Button"/>
  </mx:AddChild>
  ```

- Setting or changing any of the following component characteristics:

  Properties, by using the SetProperty class. The following line, for example, disables the button1 Button control.

  ```
  <mx:SetProperty target="{button1}" name="enabled" value="false"/>
  ```

Styles, by using the SetStyle class. The following line, for example, sets the `color` property of the application or custom component:

```
<mx:SetStyle name="color" value="0xAAAAAA"/>
```

Event listeners, by using the SetEventHandler class. The following line, for example, sets the `click` event listener of the button1 Button control:

```
<mx:SetEventHandler target="{button1}" name="click"
  handler="newClickHandler()"/>
```

- Using a custom class that you define by implementing the IOverride interface. For information on creating and using custom overrides, see "Creating your own override classes" on page 1048.

Example: Login form application

The following example creates the Login and Register forms shown in "Example: Login form application" on page 1026. This application has the following features:

- When the user clicks the Need to Register LinkButton control, the event handler for the `click` event sets the view state to Register.
- The Register state code adds a TextInput control, changes properties of the Panel container and Button control, removes the existing LinkButton controls, and adds a new LinkButton control.

- When the user clicks the Return to Login LinkButton control, the event handler for the `click` event resets the view state to the base view state.

```
<?xml version="1.0"?>
<!-- states\LoginExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    verticalAlign="middle">

    <!-- The Application class states property defines the view states.-->
    <mx:states>
        <mx:State name="Register">
            <!-- Add a TextInput control to the form. -->
            <mx:AddChild relativeTo="{loginForm}"
                position="lastChild"
                creationPolicy="all">
                <mx:FormItem id="confirm" label="Confirm:">
                    <mx:TextInput/>
                </mx:FormItem>
            </mx:AddChild>

       <!-- Set properties on the Panel container and Button control.-->
            <mx:SetProperty target="{loginPanel}"
                name="title" value="Register"/>
            <mx:SetProperty target="{loginButton}"
                name="label" value="Register"/>

            <!-- Remove the existing LinkButton control.-->
            <mx:RemoveChild target="{registerLink}"/>

            <!-- Add a new LinkButton control to change view state
                back to the login form.-->
            <mx:AddChild relativeTo="{spacer1}" position="before">
                <mx:LinkButton label="Return to Login"
                    click="currentState=''"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:Panel id="loginPanel"
        title="Login"
        horizontalScrollPolicy="off"
        verticalScrollPolicy="off">
        <mx:Form id="loginForm">
            <mx:FormItem label="Username:">
                <mx:TextInput/>
            </mx:FormItem>
            <mx:FormItem label="Password:">
                <mx:TextInput/>
            </mx:FormItem>
        </mx:Form>
```

```
        <mx:ControlBar>
      <!-- Use the LinkButton to change to the Register view state.-->
            <mx:LinkButton id="registerLink"
                label="Need to Register?"
                click="currentState='Register'"/>
            <mx:Spacer width="100%" id="spacer1"/>
            <mx:Button label="Login" id="loginButton"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

Creating a view state in ActionScript

To create a state in ActionScript, you do the following:

1. Import the classes of the mx.states package.
2. Declare a State variable that is available to the code that will use it.
3. Create a function that does the following
   a. Instantiate a new State object
   b. Assign the object to the variable you defined in step 1.
   c. Set the state name.
   d. Declare variables for the override class objects and assign them to new instances of the classes (such as SetProperty).
   e. Specify the properties of the override instances
   f. Add the override instances to the State object's `overrides` array.
   g. Add the state to the `states` array.
4. Call the function as part of an application or custom component initialization event listener, typically in response to the `initialize` event.

The following example uses ActionScript to creates a view state that sets a style property of an HBox container:

```
<?xml version="1.0"?>
<!-- states\StatesAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="createState();" >

    <mx:Script>
        <![CDATA[

            import mx.states.*;

            // Variable for the ActionScript-defined view state.
            private var newState:State;

            // Initialization method and
            // creates a view state using ActionScript.
            private function createState():void {
                newState = new State();
                newState.name = "state1";
                var myProp:SetStyle = new SetStyle();
                myProp.name="backgroundColor";
                myProp.value= "#CCCCCC";
                myProp.target = myHB;
                newState.overrides[0] = myProp;
                states.push(newState);
            }
        ]]>
    </mx:Script>

    <mx:HBox id="myHB" height="50%" width="50%">
        <mx:Button label="Base state" click="currentState='';"/>
        <mx:Button label="Change state" click="currentState='state1';"/>
    </mx:HBox>
</mx:Application>
```

Setting component properties

You can use the SetProperty class to specify a property value that is in effect only during a specific view state. For example, the following code sets properties of a Panel container and of a Button control when you switch to the Register view state:

```
<?xml version="1.0"?>
<!-- states\StatesSetProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="Register">
            <mx:SetProperty
                target="{loginPanel}"
                name="title"
                value="Register"/>
            <mx:SetProperty
                target="{loginButton}"
                name="label"
                value="Register"/>
        </mx:State>
    </mx:states>

    <mx:Panel id="loginPanel"
        title="Login"
        horizontalScrollPolicy="off"
        verticalScrollPolicy="off">

        <mx:Form id="loginForm">
            <mx:Button label="Login" id="loginButton"/>
        </mx:Form>

        <mx:ControlBar width="100%">
            <mx:Button label="Change State"
              click="currentState =
                currentState=='Register' ? '':'Register';"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

You can use data binding to specify information to the `value` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- states\StatesSetProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define a variable in ActionScript.
            [Bindable]
            public var registerValue:String="Register";
        ]]>
    </mx:Script>

    <mx:states>
        <mx:State name="Register">
            <mx:SetProperty
                target="{loginPanel}"
                name="title"
                value="{registerValue}"/>
            <mx:SetProperty
                target="{loginButton}"
                name="label"
                value="{registerValue}"/>
        </mx:State>
    </mx:states>

    <mx:Panel id="loginPanel"
        title="Login"
        horizontalScrollPolicy="off"
        verticalScrollPolicy="off">

        <mx:Form id="loginForm">
            <mx:Button label="Login" id="loginButton"/>
        </mx:Form>

        <mx:ControlBar width="100%">
            <mx:Button label="Change State"
              click="currentState =
                currentState=='Register' ? '':'Register';"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

When you switch to the Register state, the `value` property of the SetProperty class is determined by the registerValue variable. However, this binding occurs only when you switch to the Register view state; if you modify the value of the registerValue variable after switching to the Register view state, the `title` property of the target component is not updated.

Adding and removing components

The AddChild and RemoveChild classes let you add or remove child components. When you add a component in ActionScript, you must construct the component before you use the AddChild class. Removing a child does not delete it, so you can redisplay it later without recreating it.

Using the AddChild and RemoveChild classes

When you add a child, you can use the `relativeTo` property to specify the container relative to which you are placing the new child; the default value is the application or custom component that defines the view state. For example, to add a Button control to a HBox container, you specify the container as the value of the `relativeTo` property.

You can use the `position` property to specify the child's location within the container. valid values are `before`, `after`, `firstChild`, and `lastChild`. The default value is `lastChild`.

The following example adds a Button control as the first child of an HBox container named h1:

```
<?xml version="1.0"?>
<!-- states\StatesAddRelative.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewButton">
            <mx:AddChild relativeTo="{h1}" position="firstChild">
                <mx:Button id="buttonNew" label="New Button"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:HBox id="h1">
        <mx:Button label="Change State"
            click="currentState =
                currentState=='NewButton' ? '':'NewButton';"/>
    </mx:HBox>
</mx:Application>
```

You cannot use view states to directly change the parent of a component. For example, if you want to move a component from one container to another, you have to use the RemoveChild class to remove the child from its parent container, and then use the AddChild class to add the component to a different container, as the following example shows:

```
<?xml version="1.0"?>
<!-- states\StatesAddRemove.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewParent">
            <mx:RemoveChild target="{button1}"/>
            <mx:AddChild target="{button1}" relativeTo="{v2}"/>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1" borderStyle="solid">
        <mx:Label text = "VBox v1"/>
        <mx:Button id="button1"/>
    </mx:VBox>

    <mx:VBox id="v2" borderStyle="solid">
        <mx:Label text = "VBox v2"/>
        <mx:Button label="Change Parent"
            click="currentState =
                currentState=='NewParent' ? '':'NewParent';"/>
    </mx:VBox>
</mx:Application>
```

Controlling when to create added children

By default, Flex creates children when they are first required, so the application loads as quickly as possible. However, if a child requires a long time to create, users might see a delay, when the state changes. Therefore, you might create the child before the state changes to improve your application's apparent speed and responsiveness. Flex lets you choose among three ways of creating a child:

- Automatically create it when it is first needed (the default).
- Automatically create it when the application first loads.
- Have the application explicitly create the child.

The specification of when the child is created is the *creation policy.* The following section describes how you determine the creation policy of children created by view states. For more general information on creation policies and controlling how children are created, see Chapter 6, "Improving Startup Performance," in *Building and Deploying Flex 2 Applications*.

About the AddChild child specification properties

The AddChild class has two mutually exclusive properties that you use to specify the child class to add:

target Specifies directly the component to add. When you use this property, Flex creates the child when the applications starts. You cannot use any other creation policy with this property. Using this property is equivalent to specifying a Flex component as the `targetFactory` property and using the `all` creation policy.

targetFactory Specifies either of the following items:

- A factory class that implements the IDeferredInstance interface and creates the child instance or instances.
- A Flex component, (that is, any class that is a subclass of the UIComponent class), such as the Button control. If you use a Flex component, the Flex compiler automatically wraps the component in a factory class.

About the targetFactory property

The `targetFactory` property is the default property of the AddChild class, so if you specify the child by using an MXML tag in the `<mx:AddChild>` tag body, as shown in the following code example, Flex automatically uses the `targetFactory` property and creates the required factory class:

```
<mx:AddChild relativeTo="{v1}">
  <mx:Button id="b0" label="New Button" />
</mx:AddChild>
```

The `targetFactory` property lets you specify the creation policy for the child component. You use the following property and method of the AddChild class with a `targetFactory` property:

creationPolicy property Specifies when to create the child, by using the following values:

- `auto` Creates the instance when it is first added. This is the default value. In a script, specify this property by using the `mx.core.ContainerCreationPolicy.AUTO` constant.
- `all` Creates the instance at application startup. In a script, specify this property by using the `ContainerCreationPolicy.ALL` constant.
- `none` Requires your application to call the AddChild class `createInstance` method to create an instance of the child. In a script, specify this property by using the `mx.core.ContainerCreationPolicy.NONE` constant.

createInstance() method Uses the factory class to create an instance of the child. If you specify a `creationPolicy` value of `none`, call this method to create the child instance. You can call the `createInstance()` method with any creation policy, but if the child has already been created, it does nothing. For example, if you have a `creationPolicy` value of `auto`, but need to access a child before the state is entered, you can call the `createInstance()` method to ensure that the child has been created.

The following example adds a child and has Flex create a child instance when it first starts up. The application does not display the child until the state with the AddChild tag is activated.

```
<mx:AddChild relativeTo="{v1}" position="lastChild" creationPolicy="all">
  <mx:Button id="b0" label="New Button"/>
</mx:AddChild>
```

Example: Dynamically adding a Button control.

The following example lets you change between three states, a base state, a state that adds a statically created Button control, and a state that adds a Button control that is not created until the state is entered.

```
<?xml version="1.0"?>
<!-- states\StatesDefInstan.mxml  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initializeApplication();">

    <mx:Script>
        <![CDATA[
            import mx.core.*;
            import mx.states.*;

            private function initializeApplication():void {
                //Code for adding a static button.
                // Create the Button to be added
                var newButton:Button = new Button();
                newButton.label = "New Button";

                // Create an AddChild object that adds the button.
                var addChild:AddChild = new AddChild();
                addChild.relativeTo = myPanel;
                addChild.target = newButton;

                // Create a State object
                var newState:State = new State();
                newState.name = "added";
                newState.overrides = new Array();
                newState.overrides.push(addChild);

                // Initialize the states property of the Application.
                // and add the new state.
                states = new Array ();
                states.push(newState);

                //Code for adding a dynamically created button.
                // Create an add AddChild object that adds the button.
                var addChildDyn:AddChild = new AddChild();
                addChildDyn.relativeTo = myPanel;

                // Explicitly set the AddChild targetFactory property
                // to a factory
               // that runs the createMyButton method to create the button.
                addChildDyn.targetFactory =
                    new DeferredInstanceFromFunction(createMyButton);

                // Create State object with the addChildDyn override.
```

```
                var newState2:State = new State();
                newState2.name = "dynAdded";
                newState2.overrides = new Array();
                newState2.overrides.push(addChildDyn);

                //Add the state to the Application's states property.
                states.push(newState2);
            }

            // Function to create a new Button control.
            public function createMyButton():Object {
                var newButton:Button = new Button();
                newButton.label = "New Dynamic Button";
                return newButton;
            }

        ]]>
    </mx:Script>
    <mx:Panel id="myPanel"
        title="Static and dynamic states"
        width="300" height="150">
        <!-- If the current state is not the added state, change to the
            added state; if the current state is the added state,
            change to the base state. -->
        <mx:Button id="myButton" label="Toggle Static Button"
           click="currentState = currentState == 'added' ? '' : 'added';"/>
        <!-- If the current state is not the dynAdded state, change to the
            dynAdded state; if current state is the dynAdded state,
            change to the base state. -->
        <mx:Button id="myButton2"
            label="Toggle Dynamic Button"
            click="currentState =
                currentState == 'dynAdded' ? '' : 'dynAdded';"/>
    </mx:Panel>
</mx:Application>
```

Setting event listeners

Flex lets you define state-specific event listeners in your application. A state-specific event listener is only active during a specific view state. For example, you might define a Button control that uses one event listener in the base view state, but uses a different event handler when you change view state.

You can set state-specific event listeners in either of two ways:

- In MXML you can use the `handler` event type of the SetEventHandler class to specify the event listener.

- In ActionScript or MXML, you can use the `handlerFunction` property of the SetEventHandler class to specify the event listener.

Using the handler event type

The `handler` event type has the following advantages over the `handlerFunction` property:

- The event listener can take multiple parameters. If you use the `handlerFunction` property, the handler function can take only the event object as a parameter.
- You can specify ActionScript code directly in the `<mx:SetEventHandler>` tag, without having to define a separate event listener function. This technique is useful for very simple event listeners, such as when you want to pop up an Alert dialog box.

You can use the `handler` event type in MXML only. When you use the `handler` event type, you use the `name` property to specify the event name.

The following example shows how to use the `handler` property to specify the event listener code directly in the `<mx:SetEventHandler>` tag for the `click` event of a Button control:

```
<mx:SetEventHandler target="{myButton1}" name="click"
  handler="Alert.show('Hello World.')"/>
```

The following line shows how you specify a handler function that takes two parameters, the event, and a user ID variable.

```
<mx:SetEventHandler target="{myButton1}" name="click"
  handler="myAlert(event, userID)"/>
```

Using the handlerFunction property

You can use the `handlerFunction` property in ActionScript or MXML. It lets you specify an event listener function that takes a single flash.events.Event attribute. The following example code sets an event listener:

```
<mx:SetEventHandler target="{myButton1}" name="click"
  handlerFunction="handler2"/>
```

The handler2 event listener must have a declaration, as in the following line:

```
private function handler2(theEvent:Event)
```

Example: Setting event listeners

The following example creates the base view state and two additional view state. Each state uses the SetEventHandler class to define a different event listener for the `click` event for the Button control named b1.

Buttons controls below the text area let you select the current state, including the base state. The event handler for each view state displays information about the state, and the value of any second parameter that was passed to the event listener.

```
<?xml version="1.0"?>
<!-- states\StatesEventHandlersSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    currentState="MXMLState2">

    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            // Functions for setting click event listeners.
            private function handlerBase(theEvent:Event,
                theString:String):void {
                ta1.text = "Default click handler" +
                    "\nParameter 2: " + theString;
            }

            private function handlerMXMLState2(theString:String):void {
                ta1.text = "MXML state handler function" +
                    "\nParameter 2: " + theString;
            }
        ]]>
    </mx:Script>

    <!-- The Application's states property defines two view states. -->
    <mx:states>
        <mx:State name="MXMLState1">
            <mx:SetEventHandler name="click" target="{b1}"
                handler="ta1.text='MXML-inline event listener \n
                    No second parameter';"/>
        </mx:State>

        <!-- The mxmlState2 State specifies a click event listener
            that takes a second parameter. -->
        <mx:State name="MXMLState2">
            <mx:SetEventHandler name="click" target="{b1}"
              handler="handlerMXMLState2('MXML-defined event listener');"/>
        </mx:State>
    </mx:states>

    <mx:Button id="b1"
        label="Click Me"
        click="handlerBase(event, 'Hi there');"/>
    <mx:TextArea id="ta1" height="100" width ="50%"/>
    <mx:HBox>
        <mx:Button id="b2"
            label="MXML state, inline handler"
```

```
            click="currentState='MXMLState1';"/>
        <mx:Button id="b3"
            label="MXML state, handler function"
            click="currentState='MXMLState2';"/>
        <mx:Button id="b4"
            label="Base state"
            click="currentState='';"/>
    </mx:HBox>
</mx:Application>
```

Using view state events

When a component's `currentState` property changes, the State object for the states being exited and entered dispatch the following events:

enterState Dispatched when a view state is entered, but not fully applied. Dispatched by a State object after it has been entered, and by a component after it returns to the base view state.

exitState Dispatched when a view state is about to be exited. It is dispatched by a State object before it is exited, and by a component before it exits the base view state.

The component on which you modify the `currentState` property to cause the state change dispatches the following events:

currentStateChanging Dispatched when the view state is about to change. It is dispatched by a component after its `currentState` property changes, but before the view state changes. You can use this event to request any data from the server required by the new view state.

currentStateChange Dispatched after the view state has completed changing. It is dispatched by a component after its `currentState` property changed. You can use this event to send data back to a server indicating the user's current view state.

Example: Using view states with history management

The Flex History Manager lets users navigate through a Flex application by using the web browser's back and forward navigation commands. The History Manager can track when the application enters a state so that users can use the browser to navigate between states, such as states that correspond to different stages in an application process.

To enable history management to track application states, your application must do the following:

- Implement the IHistoryState interface by defining `loadState` and `saveState` methods.
- Register the application with the History Manager.
- Each time the state changes, call the `HistoryManager.save` method.

The following code shows how you can code a search interface so that the History Manager keeps track of your search. For more information on history management, including another example of tracking states, see Chapter 32, "Using the History Manager," on page 1147.

```
<?xml version="1.0"?>
<!-- states\StatesHistoryManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    implements="mx.managers.IHistoryManagerClient"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
        import mx.managers.HistoryManager;

        ////////////////////////
        // IHistoryState methods
        ////////////////////////
        // Restore the state and searchString value.
        public function loadState(state:Object):void {
            if (state) {
                currentState = state.currentState;
                searchString = searchInput.text = state.searchString;
            }
            else {
                currentState = '';
            }
        }
        // Save the current state and the searchString value.
        public function saveState():Object {
            var state:Object = {};
            state.currentState = currentState;
            state.searchString = searchString;
            return state;
        }

        ////////////////////////
        // App-specific scripts
        ////////////////////////
        // The search string value.
        [Bindable]
        public var searchString:String;

        // Register the application with the history manager
        // when the application is created.
        public function initApp():void {
            HistoryManager.register(this);
        }

        // The method for doing the search.
        // For the sake of simplicity it doesn't do any searching.
```

```
        // It does change the state to display the results box,
        // and save the new state in the history manager.
        public function doSearch():void {
            currentState = "results";
            searchString = searchInput.text;
            HistoryManager.save();
        }
        // Method to revert the state to the base state.
        // Saves the new state in the history manager.
        public function reset():void {
            currentState = '';
            searchInput.text = "";
            searchString = "";
            HistoryManager.save();
        }
        ]]>
    </mx:Script>

    <mx:states>
        <!-- The state for displaying the search results -->
        <mx:State name="results">
            <mx:SetProperty target="{p}" name="width" value="100%" />
            <mx:SetProperty target="{p}" name="height" value="100%" />
            <mx:SetProperty target="{p}" name="title" value="Results" />
            <mx:AddChild relativeTo="{searchFields}">
                <mx:Button label="Reset" click="reset()" />
            </mx:AddChild>
            <mx:AddChild relativeTo="{p}">
                <mx:Label text="Search results for {searchString}" />
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <!-- In the base state, just show a panel
        with a search text input and button. -->
    <mx:Panel id="p" title="Search" resizeEffect="Resize">
        <mx:HBox id="searchFields" defaultButton="{b}">
            <mx:TextInput id="searchInput" />
            <mx:Button id="b" label="Go" click="doSearch();" />
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```

Example: Using view states with a custom item renderer

The following code shows an application that uses a custom item renderer to display catalog items. When the user moves the mouse over an item, the item renderer changes to a state where the picture is slightly enlarged, the price appears in the cell, and the application's text box shows a message about the item. All changes are made by the item renderer's state, including the change in the parent application.

```
<?xml version="1.0"?>
<!-- states\StatesRendererMain.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="450" height="500">
    <mx:Script>
        <![CDATA[
            import mx.controls.listClasses.*;
            import mx.controls.Image;
            import mx.collections.ArrayCollection;

            //The data to display in the TileList control.
            [Bindable]
            public var catArray:ArrayCollection = new ArrayCollection([
                { name: "USB Watch",
                  data: "1",
                  price: "129.99",
                  image: "assets/usbwatch.jpg",
                  description: "So, you need to tell the time of course" },
                { name: "007 Digital Camera",
                  data: "1",
                  price: "99.99",
                  image: "assets/007camera.jpg",
                  description: "Just like 007 used" },
                { name: "2-Way Radio Watch",
                  data: "1",
                  price: "49.99",
                  image: "assets/radiowatch.jpg",
                  description: "Better than Dick Tracy's" },
                { name: "USB Desk Fan",
                  data: "1",
                  price: "19.99",
                  image: "assets/usbfan.jpg",
                  description: "Computer-powered cool!!!"},
            ]);
        ]]>
    </mx:Script>

    <mx:TileList id="myList"
        dataProvider="{catArray}"
        columnWidth="150"
```

```
        rowHeight="150"
        width="310"
        height="310"
        itemRenderer="myComponents.ImageComp"/>

    <!-- The item renderer fills in the TextArea control. -->
    <mx:HBox>
        <mx:Label text="Description"/>
        <mx:TextArea id="t1"
            width="200" height="100"
            wordWrap="true"/>
    </mx:HBox>
</mx:Application>
```

The following code defines the item renderer, in the file imagecomp.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- states\myComponents\ImageComp.mxml
    When the mouse pointer goes over a cell,
        this component changes its state to showdesc.
    When the mouse pointer goes out of the cell,
        the component returns to the base state. -->

<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="center"
    verticalAlign="top"
    mouseOver="currentState='showdesc'"
    mouseOut="currentState=''">

    <!-- In the base state, display the image and the label. -->
    <mx:Image id="img1"
        source="{data.image}"
        width="75"
        height="75"/>
    <mx:Label text="{data.name}"/>

    <mx:states>
        <mx:State name="showdesc">
          <!-- In the showdesc state, add the price, make the image bigger,
            and put the description in the parent application's TextArea.-->
            <mx:AddChild>
                <mx:Text text="{data.price}"/>
            </mx:AddChild>
            <mx:SetProperty target="{img1}" name="width" value="85"/>
            <mx:SetProperty target="{img1}" name="height" value="85"/>
            <mx:SetProperty target="{parentApplication.t1}" name="text"
                value="{data.description}"/>
        </mx:State>
    </mx:states>
</mx:VBox>
```

Building applications by using view states

You can use the following features and techniques to structure and build efficient, feature-rich, view state–based components and applications:

Using view states in custom components

If one or more view states apply to a single component or set of components, you define a custom component that comprises this component or components, and specify the view states in the component definition, rather than at the Application tag level. For example, if a view state adds a Button control to an HBox container, consider making the HBox a custom component and defining a state in the component that adds the button. You should consider doing this particularly if you have multiple states that do nothing but modify the HBox container.

Similarly, if a custom component has multiple view states, define the view states in the component code, not in the main application. For example, if a custom TitleWindow component has a collapsed view state and a expanded view state, you should define these view states in the panel MXML component, not in the main application file.

This way, you segregate the view state definitions into the specific components to which they apply. A item renderer is one example of good use of a multiple view state MXML component.

The following example shows a custom component for a TitleWindow component that has two view states, collapsed and expanded:

```
<?xml version="1.0"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
    close="checkCollapse();" headerHeight="40" showCloseButton="true">

    <mx:Script>
        <![CDATA[

            // Skins for the close button when the
            // TitleWindow container is collapsed.
            [Embed(source="closeButtonUpSkin.jpg")]
            [Bindable]
            public var closeBUp:Class;

            [Embed(source="closeButtonDownSkin.jpg")]
            [Bindable]
            public var closeBDown:Class;

            [Embed(source="closeButtonOverSkin.jpg")]
            [Bindable]
            public var closeBOver:Class;

            private function checkCollapse():void {
                currentState =
                    currentState == "collapsed" ? "" : "collapsed";
            }
    ]]>
    </mx:Script>

    <mx:states>
        <mx:State name="collapsed">
            <mx:SetProperty
                name="height"
                value="{getStyle('headerHeight')}"/>
            <mx:SetStyle
                name="closeButtonUpSkin"
                value="{closeBUp}"/>
            <mx:SetStyle
                name="closeButtonDownSkin"
                value="{closeBDown}"/>
            <mx:SetStyle
                name="closeButtonOverSkin"
                value="{closeBOver}"/>
        </mx:State>
    </mx:states>
</mx:TitleWindow>
```

This example replaces the default close icon for a TitleWindow when the component is in the collapsed state.

You can then use this component in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- states\StatesTitleWindowMain.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateTitleWindow id="myP" title="My Application">
        <mx:HBox width="100%">
            <mx:Button/>
            <mx:TextArea/>
        </mx:HBox>

        <mx:ControlBar width="100%">
            <mx:Label text="Quantity"/>
            <mx:NumericStepper/>
            <!-- Use Spacer to push Button control to the right. -->
            <mx:Spacer width="100%"/>
            <mx:Button label="Add to Cart"/>
        </mx:ControlBar>
    </MyComp:StateTitleWindow>
</mx:Application>
```

Using a non-base initial view state

By default, a Flex application starts in the base view state. The base view state corresponds to the `currentState` property set to `""`.

However, an application or component can set the initial view state to a non-base view state. In a search interface component, for example, the most commonly used interface might be the initial expanded view, not the collapsed view.

To use a view state other than the base state as the initial view state, set the `currentState` property to the specific view state. Use code such as the following, for example, to specify that a component is initially in its collapsed state:

```
<mx:VBox currentState="collapsed">
```

Additional state-based application techniques

Consider the following additional techniques for structuring and implementing a state-based applications:

- View states are one way for you to change the appearance of a control in response to a user action. You can also use navigator containers, such as the Accordion, Tab Navigator, and ViewStack containers when you perform changes that affect several components. For example, you might use the Accordion container to let the user navigate through a complex form, rather than using view states to perform this action. For more information on navigator containers, see Chapter 16, "Using Navigator Containers," on page 627.
- Create cascading view state definitions, where you use the `basedOn` property to explicitly base one view state on another view state. Use this technique when you have multiple view states that all include some common elements. You separate all the common elements into one view state on which you base the other view states. The drill-down search interface described in "Use case: Login interface" on page 1020 uses this technique.
- Use view states that replace major sections of the display. For example, a shopping application could use view states to control whether the main part of the display shows a product selector panel or a checkout accordion.
- Use transitions to control the changes between view states; for example, you can use the following code to add a resize effect to any component that changes size when the view state changes. For more information on transitions, see Chapter 28, "Using Transitions," on page 1051.

Creating your own override classes

You can define custom overrides by creating a class that implements the IOverride interface. For example, you can create a class that adds a bitmap filter, such as a filter that blurs an object.

The IOverride interface contains the following methods:

Method	Description
`initialize()`	Initializes the override.
`apply()`	Saves the original value and applies the override.
`remove()`	Restores the original value.

The following example shows an AddBlur override class that applies the Flash BlurFilter class to blur the target component.

```
package myOverrides
```

```
{
  import flash.display.*;
  import flash.filters.*;
  import mx.core.*;
  import mx.states.*;

  /* State override that adds a Blur effect to a component. */
  public class AddBlur implements IOverride
  {
    /* Constructor. */
    public function AddBlur(
      target:DisplayObject = null)
    {
      this.target = target;
    }

    /* Property: The object to blur. */
    public var target:DisplayObject;

    /* The initialize() method is empty for this example. */
    public function initialize():void {
    }

    /* The apply method adds a BlurFilter to the filters array. */
    public function apply(parent:UIComponent):void {
      var obj:DisplayObject = target ? target : parent;
      var filters:Array = obj.filters;

      filters.push(new BlurFilter());
      obj.filters = filters;
    }

    /* The remove method removes the BlurFilter from the filters array. */
    public function remove(parent:UIComponent):void {
      var obj:DisplayObject = target ? target : parent;
      var filters:Array = obj.filters;

      filters.pop();
      obj.filters = filters;
    }
  }
}
```

CHAPTER 28

Using Transitions

28

View states let you change appearance of an application, typically in response to a user action. Transitions define how a change of view state looks as it occurs on the screen. You define a transition by using the effect classes, in combination with several effects designed explicitly for handling transitions. This topic describes how to define transitions to work with view states.

To use transitions, you should be familiar with how effects and view states work. For more information on effects, see Chapter 17, "Using Behaviors," on page 649. For more information on view states, see Chapter 27, "Using View States," on page 1019.

Contents

About transitions

View states let you vary the content and appearance of an application, typically in response to a user action. To use view states, you define the base view state of an application, and one or more additional view states that specify modifications to the base view state.

When you change view states, Adobe Flex performs all the visual changes to the application at the same time. That means when you resize, move, or in some other way alter the appearance of application components, the components appear to jump from one state to the next.

Instead, you might want to define a smooth visual change from one state to the next, in which the change occurs over a period of time. A *transition* is one or more effects grouped together to play when a view state change occurs.

For example, your application defines a form that in its base view state shows only a few fields, but in an expanded view state shows additional fields. Rather than jumping from the base version of the form to the expanded version, you define a Flex transition that uses the Resize effect to expand the form, and then uses the Fade effect to slowly make the new form elements appear on the screen.

When a user changes back to the base version of the form, your transition uses a Fade effect to make the fields of the expanded form disappear, and then uses the Resize effect to slowly shrink the form back to its original size.

By using a transition, you can apply one or more effects to the form, to the fields added to the form, and to fields removed from the form. You can apply the same effects to each form field, or apply different effects. You can also define one set of effects to play when you change state to the expanded form, and a different set of effects to play when changing from the expanded state back to the base state.

Comparing transitions to effects

Transitions do not replace effects; that is, you can still apply a single effect to a component, and invoke that effect by using an effect trigger, or the `playEffect()` method.

Transitions give you the ability to group multiple effects so that the effects are triggered together as part of a change of view states. Transitions are designed to work specifically with a change to the current view state, because a change in view state change typically means multiple components are modified at the same time.

Transitions also support effect filters. A *filter* lets you conditionalize an effect so that it plays an effect target only when the target changes in a certain way. For example, as part of the change to the view state, one or more components may change size. You can use a filter with the Blur effect so that you apply the Blur effect only to the components being resized.

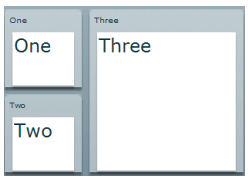
Defining transitions

You use the Transition class to create a transition. The following table defines the properties of the Transition class:

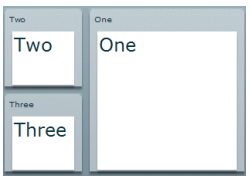
Property	Definition
`fromState`	A String that specifies the view state that you are changing from when you apply the transition. The default value is an asterisk, `"*"`, which means any view state.
`toState`	A String that specifies the view state that you are changing to when you apply the transition. The default value is an asterisk, `"*"`, which means any view state.
`effect`	The Effect object to play when you apply the transition. Typically, this is a composite effect, such as the Parallel or Sequence effect, that contains multiple effects.

You can define multiple Transition objects in an application. The UIComponent class includes a `transitions` property that you set to an Array of Transition objects.

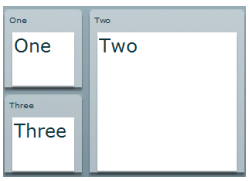
You could define an application with three Panel containers and three view states, as the following example shows:



Base view state



One view state



Two view state

You define the transitions for this example in MXML using the <mx:transitions> tag, as the following example shows:

```
<?xml version="1.0" ?>
<!-- transitions/DefiningTrans.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="400" >

    <!-- Define the two view states, in addition to the base state.-->
    <mx:states>
        <mx:State name="One">
            <mx:SetProperty target="{p1}" name="x" value="110"/>
            <mx:SetProperty target="{p1}" name="y" value="0"/>
            <mx:SetProperty target="{p1}" name="width" value="200"/>
            <mx:SetProperty target="{p1}" name="height" value="210"/>
            <mx:SetProperty target="{p2}" name="x" value="0"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="100"/>
            <mx:SetProperty target="{p2}" name="height" value="100"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
        <mx:State name="Two">
            <mx:SetProperty target="{p2}" name="x" value="110"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="200"/>
            <mx:SetProperty target="{p2}" name="height" value="210"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
    </mx:states>

    <!-- Define Transition array with one Transition object.-->
    <mx:transitions>
        <!-- Define a transition for changing from any state to any state.
        -->
        <mx:Transition id="myTransition" fromState="*" toState="*">
            <!-- Define a Parallel effect as the top-level effect.-->
            <mx:Parallel id="t1" targets="{[p1,p2,p3]}">
                <!-- Define a Move and Resize effect.-->
                <mx:Move  duration="400"/>
                <mx:Resize duration="400"/>
            </mx:Parallel>
        </mx:Transition>
    </mx:transitions>

    <!-- Define the Canvas container holdig the three Panel containers.-->
    <mx:Canvas id="pm" width="100%" height="100%" >
```

```
        <mx:Panel id="p1" title="One"
                x="0" y="0" width="100" height="100"
                click="currentState='One'" >
            <mx:Label fontSize="24" text="One"/>
        </mx:Panel>

        <mx:Panel id="p2" title="Two"
                x="0" y="110" width="100" height="100"
                click="currentState='Two'" >
            <mx:Label fontSize="24" text="Two"/>
        </mx:Panel>

        <mx:Panel id="p3" title="Three"
                x="110" y="0" width="200" height="210"
                click="currentState='' " >
            <mx:Label fontSize="24" text="Three"/>
        </mx:Panel>
    </mx:Canvas>
</mx:Application>
```

You use the `click` event of each Panel container to change the view state. When the application changes view state, Flex searches for a Transition object that matches the current and destination view state. In this example, you set the `fromState` and `toState` properties to `"*"`. Therefore, Flex applies myTransition to all state changes. For more information on setting the `fromState` and `toState` properties, see "Defining multiple transitions" on page 1056.

After Flex determines the transition that matches the change of view state, Flex applies the Move and Resize effects defined by the transition to the effect targets. In this example, you use the specification of the top-level Parallel effect in the transition to specify the targets as the three Panel containers. For more information on setting the effect targets, see "Defining effect targets" on page 1057.

The Move and Resize effects play in parallel on all effect targets, so the three Panel containers move to a new position and change size simultaneously. You can also define the top-level effect as a Sequence effect to make the Move and Resize effects occur sequentially, rather than in parallel.

Flex determines the start and end property values of the Move and Resize effect by using information from any properties that you specified to the effect, the current view state, and the destination view state. In this example, you omit any property specifications in the effect definitions, so Flex uses the current size and position of each panel container to determine the values of the `Move.xFrom`, `Move.yFrom`, `Resize.widthFrom`, and `Resize.heightFrom` properties. Flex uses the destination view state to determine the values of the `Move.xTo`, `Move.yTo`, `Resize.widthTo`, and `Resize.heightTo` properties. For more information, see "Defining the effect start and end values" on page 1057.

Defining multiple transitions

You can define multiple transitions in your application so that you can associate a specific transition with a specific change to the view state. To specify the transition associated with a change to the view states, you use the `fromState` and `toState` properties.

By default, both the `fromState` and `toState` properties are set to `"*"`, which indicates that the transition should be applied to any changes in the view state. You can set either property to an empty string, `""`, which corresponds to the base view state.

You use the `fromState` property to explicitly specify the view state that your are changing from, and the `toState` property to explicitly specify the view state that you are changing to, as the following example shows:

```
<mx:transitions>
  <!-- Play for a change to the login view state from any view state. -->
  <mx:Transition id="toLoginFromAny" fromState="*" toState="login">
    ...
  </mx:Transition>

  <!-- Play for a change to the login view state from
    the details view state. -->
  <mx:Transition id="toLoginFromDetails"
    fromState="details" toState="login">
    ...
  </mx:Transition>

  <!-- Play for a change from any view state to any other view state. -->
  <mx:Transition id="toAnyFromAny" fromState="*" toState="*">
    ...
  </mx:Transition>
</mx:transitions>

<!-- Go to the login view state, transition toLoginFromAny plays. -->
<mx:Button click="currentState="login";/>

<!-- Go to the details view state, transition toAnyFromAny plays. -->
<mx:Button click="currentState="details";/>

<!-- Go to the login view state, transition toLoginFromDetails plays
  because you transitioned from the details to the login view state. -->
<mx:Button click="currentState="login";/>

<!-- Go to the base view state, transition toAnyFromAny plays. -->
<mx:Button click="currentState='';/>
```

If a state change matches two transitions, the `toState` property takes precedence over the `fromState` property. If more than one transition matches, Flex uses the first definition in the `transition` Array.

Defining effect targets

The `<mx:Transition>` tag shown in the section "Defining transitions" on page 1053 defines the effects that make up a transition. The top-level effect defines the target components of the effects in the transition when the effect does not explicitly define a target. In that example, the transition is performed on all three Panel containers in the application. If you want the transition to play only on the first two panels, you define the Parallel effects as the following example shows:

```
<mx:Transition fromState="*" toState="*">
  <!-- Define a Parallel effect as the top most effect.-->
  <mx:Parallel id="t1" targets="{[p1,p2]}">
    <mx:Move duration="400"/>
    <mx:Resize duration="400"/>
  </mx:Parallel>
</mx:Transition>
```

You removed the third panel from the transition, so it is no longer a target of the Move and Resize effects. Therefore, the third panel appears to jump to its new position and size during the change in view state. The other two panels show a smooth change in size and position for the 400-millisecond (ms) duration of the effects.

You can also use the `target` or `targets` properties of the effects within the transition to explicitly specify the effect target, as the following example shows:

```
<mx:Parallel id="t1" targets="{[p1,p2,p3]}">
  <mx:Move targets="{[p1,p2]}" duration="400"/>
  <mx:Resize duration="400"/>
</mx:Parallel>
```

In this example, the Resize effect plays on all three panels, while the Move effect plays only on the first two panels. You could also write this example as the following code shows:

```
<mx:Parallel id="t1" targets="{[p1,p2]}">
  <mx:Move duration="400"/>
  <mx:Resize targets="{[p1,p2,p3]}" duration="400"/>
</mx:Parallel>
```

Defining the effect start and end values

Like any effect, an effect within a transition has properties that you use to configure it. For example, most effects have properties that define starting and ending information for the target component, such as the `xFrom`, `yFrom`, `xTo`, and `yTo` properties of the Move effect.

Effects defined in a transition must determine their property values for the effect to execute. Flex uses the following rules to determine the start and end values of effect properties of a transition:

1. If the effect explicitly defines the values of any properties, use them in the transition, as the following example shows:

```
<mx:Transition fromState="*" toState="*">
  <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
    <mx:Blur duration="100"
      blurXFrom="0.0" blurXTo="10.0" blurYFrom="0.0" blurYTo="10.0"/>
   <mx:Parallel>
      <mx:Move duration="400"/>
      <mx:Resize duration="400"/>
    </mx:Parallel>
   <mx:Blur duration="100"
      blurXFrom="10.0" blurXTo="0.0" blurYFrom="10.0" blurYTo="0.0"/>
  </mx:Sequence>
</mx:Transition>
```

   In this example, the two Blur filters explicitly define the properties of the effect.

2. If the effect does not explicitly define the start values of the effect, Flex determines them from the current settings of the component, as defined by the current view state.

   In the example in rule 1, notice that the Move and Resize effects do not define start values. Therefore, Flex determines them from the current size and position of the effect targets in the current view state.

3. If the effect does not explicitly define the end values of the effect, Flex determines them from the settings of the component in the destination view state.

   In the example in rule 1, the Move and Resize effects determine the end values from the size and position of the effect targets in the destination view state. In some cases, the destination view state explicitly defines these values. If the destination view state does not define the values, Flex determines them from the settings of the base view state.

4. If there are no explicit values, and Flex cannot determine values from the current or destination view states, the effect uses its default property values.

Example: Using transitions

The example in this section shows a complete application using view states and transitions. It implements the three-panel application shown in the section "Defining transitions" on page 1053, and adds Blur effects.

```
<?xml version="1.0"?>
<?xml version="1.0" ?>
<!-- transitions\DefiningTransWithBlurs.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="400" >

    <!-- Define the two view states, in addition to the base state.-->
    <mx:states>
        <mx:State name="One">
            <mx:SetProperty target="{p1}" name="x" value="110"/>
            <mx:SetProperty target="{p1}" name="y" value="0"/>
            <mx:SetProperty target="{p1}" name="width" value="200"/>
            <mx:SetProperty target="{p1}" name="height" value="210"/>
            <mx:SetProperty target="{p2}" name="x" value="0"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="100"/>
            <mx:SetProperty target="{p2}" name="height" value="100"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
        <mx:State name="Two">
            <mx:SetProperty target="{p2}" name="x" value="110"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="200"/>
            <mx:SetProperty target="{p2}" name="height" value="210"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
    </mx:states>

    <!-- Define the single transition for all view state changes.-->
    <mx:transitions>
        <mx:Transition fromState="*" toState="*">
            <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
                <mx:Blur duration="100" blurXFrom="0.0" blurXTo="10.0"
                blurYFrom="0.0" blurYTo="10.0"/>
                <mx:Parallel>
                    <mx:Move  duration="400"/>
                    <mx:Resize duration="400"/>
                </mx:Parallel>
                <mx:Blur duration="100" blurXFrom="10.0" blurXTo="0.0"
                blurYFrom="10.0" blurYTo="0.0"/>
            </mx:Sequence>
        </mx:Transition>
    </mx:transitions>

    <!-- Define the Canvas container holding the three Panel containers.-->
```

```
    <mx:Canvas id="pm" width="100%" height="100%" >
        <mx:Panel id="p1" title="One"
                x="0" y="0" width="100" height="100"
                click="currentState='One'" >
            <mx:Label fontSize="24" text="One"/>
        </mx:Panel>

        <mx:Panel id="p2" title="Two"
                x="0" y="110" width="100" height="100"
                click="currentState='Two'" >
            <mx:Label fontSize="24" text="Two"/>
        </mx:Panel>

        <mx:Panel id="p3" title="Three"
                x="110" y="0" width="200" height="210"
                click="currentState='' " >
            <mx:Label fontSize="24" text="Three"/>
        </mx:Panel>
    </mx:Canvas>
</mx:Application>
```

Handling events when using transitions

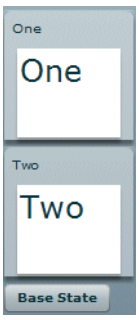
You can handle view state events, such as `currentStateChange` and `currentStateChanging`, as part of an application that defines transitions. Changes to the view state, dispatching events, and playing transition effects occur in the following order:

1. You set the `currentState` property to the destination view state.
2. Flex dispatches the `currentStateChanging` event.
3. Flex examines the list of transitions to determine the one that matches the change of the view state.
4. Flex examines the components to determine the start values of any effects.
5. Flex applies the destination view state to the application.
6. Flex dispatches the `currentStateChange` event.
7. Flex plays the effects that you defined in the transition.

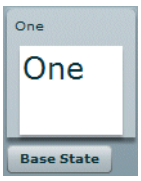
If you change state again while a transition plays, Flex jumps to the end of the transition before starting any transition associated with the new change of view state.

Using action effects in a transition

In the following example, you define an application that has two view states:



Base view state



OneOnly view state

To move from the base view state to the OneOnly view state, you create the following view state definition:

```
<mx:states>
  <mx:State name="OneOnly">
    <mx:SetProperty target="{p2}" name="visible" value="false"/>
    <mx:SetProperty target="{p2}"
      name="includeInLayout" value="false"/>
  </mx:State>
</mx:states>
```

You set the value of the `visible` and `includeInLayout` properties to `false` so that Flex makes the second Panel container invisible and ignores it when laying out the application. If the `visible` property is `false`, and the `includeInLayout` property is `true`, the container is invisible, but Flex lays out the application as if the component were visible.

A view state definition defines how to change states, and the transition defines the order in which the visual changes occur. In the example shown in the previous image, you play an Iris effect on the second panel when it disappears, and when it reappears on a transition back to the base state.

For the change from the base state to the OneOnly state, you define the toOneOnly transition which uses the Iris effect to make the second panel disappear, and then sets the panel's `visible` and `includeInLayout` properties to `false`. For a transition back to the base state, you define the toAnyFromAny transition that makes the second panel visible by setting its `visible` and `includeInLayout` properties to `true`, and then uses the Iris effect to make the panel appear, as the following example shows:

```
<mx:transitions>
  <mx:Transition id="toOneOnly" fromState="*" toState="OneOnly">
```

```
    <mx:Sequence id="t1" targets="{[p2]}">
      <mx:Iris showTarget="false" duration="350"/>
      <mx:SetPropertyAction name="visible"/>
      <mx:SetPropertyAction target="{p2}" name="includeInLayout"/>
    </mx:Sequence>
  </mx:Transition>

  <mx:Transition id="toAnyFromAny" fromState="*" toState="*">
    <mx:Sequence id="t2" targets="{[p2]}">
      <mx:SetPropertyAction target="{p2}" name="includeInLayout"/>
      <mx:SetPropertyAction name="visible"/>
      <mx:Iris showTarget="true" duration="350"/>
    </mx:Sequence>
  </mx:Transition>
</mx:transitions>
```

In the toOneOnly transition, if you hide the target by setting its `visible` property to `false`, and then play the Iris effect, you would not see the Iris effect play because the target is already invisible.

To control the order of view state changes during a transition, Flex defines several *action effects*. The previous example uses the SetPropertyAction action effect to control when to set the `visible` and `includeInLayout` properties in the transition. The following table describes the action effects:

Action effect	Corresponding view state property	Use
SetPropertyAction	`SetProperty`	Sets a property value as part of a transition.
SetStyleAction	`SetStyle`	Sets a style to a value as part of a transition.
AddChildAction	`AddChild`	Adds a child as part of a transition.
RemoveChildAction	`RemoveChild`	Removes a child as part of a transition.

When you define a view state, you can use the `SetProperty`, `SetStyle`, `AddChild`, and `RemoveChild` properties. To control when a change defined by the view state property occurs in a transition, you use the corresponding action effect. The action effects give you control over the order of the state change.

In the previous example, you used the following statement to define an action effect to occur when the value of the `visible` property of a component changes:

```
<mx:SetPropertyAction name="visible"/>
```

This action effect plays when the value of the `visible` property changes to either `true` or `false`. You can further control the effect using the `value` property of the `<mx:SetPropertyAction>` tag, as the following example shows:

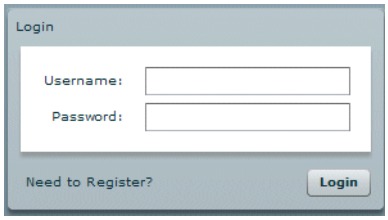
```
<mx:SetPropertyAction name="visible" value="true"/>
```

In this example, you specify to play the effect only when the value of the `visible` property changes to `true`. Adding this type of information to the action effect can be useful if you want to use filters with your transitions. For more information, see "Filtering effects" on page 1065.

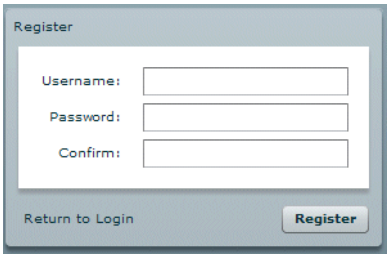
The action effects do not support a `duration` property; they only perform the specified action.

Example: Using action effects

You can define a login form with two states: a base login state and a register state, as the following example shows:



Base view state



Register view state

The view state defines how to change states, and a transition defines the order of the visual changes to the application. For example, you define the visual transition to the Register view state to occur in the following order:

1. Remove the LinkButton control to the Register form.
2. Change the title of the form to Register.
3. Change the Button label to Register.
4. Add the TextInput control to confirm the password.
5. Add the LinkButton control to return to the Login form.

When a user returns to the Login form, which means a return to the base view state, you define the visual changes to the Login form to occur in the following order:

1. Remove the LinkButton control to the Login form.
2. Remove the TextInput control to confirm the password.
3. Change the title of the form to Login.
4. Change the Button label to Login.
5. Add the LinkButton control to go to the Register form.

The following transition uses the action effects to define the visual changes when changing state from the Login form to the Register form, or when changing state from the Register form to the Login form. In this example, you use the same transition for both state changes. The effects all execute sequentially as part of the transition.

```
<?xml version="1.0" ?>
<!-- transitions\ActionTransitions.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    verticalAlign="middle">

    <!-- Define one view state, in addition to the base state.-->
    <mx:states>
        <mx:State name="Register">
            <mx:AddChild relativeTo="{loginForm}" position="lastChild">
                <mx:target>
                    <mx:FormItem id="confirm" label="Confirm:">
                        <mx:TextInput/>
                    </mx:FormItem>
                </mx:target>
            </mx:AddChild>
            <mx:SetProperty target="{loginPanel}" name="title"
            value="Register"/>
            <mx:SetProperty target="{loginButton}" name="label"
            value="Register"/>
            <mx:RemoveChild target="{registerLink}"/>
            <mx:AddChild relativeTo="{spacer1}" position="before">
                <mx:target>
                    <mx:LinkButton id="loginLink"
                        label="Return to Login"
                        click="currentState=''"/>
                </mx:target>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <!-- Define Transition array with one Transition object.-->
    <mx:transitions>
        <mx:Transition id="toRegister" fromState="*" toState="Register">
            <mx:Sequence targets="{[loginPanel, registerLink, confirm,
            loginLink, spacer1]}">
                <mx:RemoveChildAction/>
                <mx:SetPropertyAction target="{loginPanel}" name="title"/>
               <mx:SetPropertyAction  target="{loginButton}" name="label"/>
                <mx:Resize target="{loginPanel}"/>
                <mx:AddChildAction/>
            </mx:Sequence>
        </mx:Transition>
    </mx:transitions>

    <!-- Define a Panel container that defines the login form.-->
```

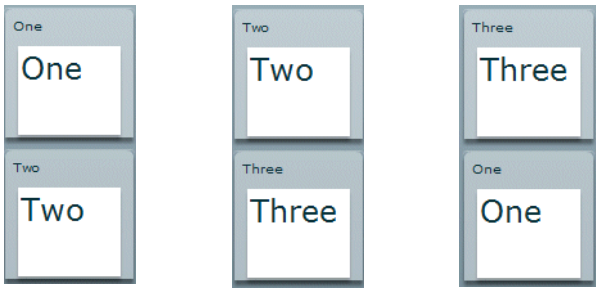
```
    <mx:Panel title="Login" id="loginPanel"
            horizontalScrollPolicy="off"
            verticalScrollPolicy="off">
        <mx:Form id="loginForm" >
            <mx:FormItem label="Username:">
                <mx:TextInput/>
            </mx:FormItem>
            <mx:FormItem label="Password:">
                <mx:TextInput/>
            </mx:FormItem>
        </mx:Form>
        <mx:ControlBar>
            <mx:LinkButton id="registerLink"  label="Need to Register?"
                click="currentState='Register'"/>
            <mx:Spacer width="100%" id="spacer1"/>
            <mx:Button label="Login" id="loginButton"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

Filtering effects

By default, Flex applies all of the effects defined in a transition to all of the target components of the transition. Therefore, in the following example, Flex applies the Move and Resize effects to all three targets:

```
<mx:Transition fromState="*" toState="*">
  <!-- Define a Parallel effect as the top most effect.-->
  <mx:Parallel id="t1" targets="{[p1,p2,p3]}">
    <!-- Define a Move and Resize effect.-->
    <mx:Move duration="400"/>
    <mx:Resize duration="400"/>
  </mx:Parallel>
</mx:Transition>
```

However, you might want to conditionalize an effect so that it does not apply to all target components, but only to a subset of the components. You can define an application with three view states, as the example shows:



Each change of view state removes the top panel, moves the bottom panel to the top, and adds the next panel to the bottom of the screen. In this example, the third panel is invisible in the base view state.

For this example, you define a single transition that applies a WipeUp effect to the top panel as it is removed, applies a Move effect to the bottom panel as it moves to the top position, and applies another WipeUp effect to the panel being added to the bottom, as the following example shows:

```
<mx:transitions>
  <mx:Transition fromState="*" toState="*">
    <mx:Sequence targets="{[p1,p2,p3]}">
      <mx:Sequence id="sequence1" filter="hide">
        <mx:WipeUp/>
        <mx:SetPropertyAction name="visible" value="false"/>
      </mx:Sequence>
      <mx:Move filter="move"/>
      <mx:Sequence id="sequence2" filter="show">
        <mx:SetPropertyAction name="visible" value="true"/>
        <mx:WipeUp/>
      </mx:Sequence>
    </mx:Sequence>
  </mx:Transition>
</mx:transitions>
```

The sequence1 Sequence effect uses the `filter` property to specify the change that a component must go through in order for the effect to play on it. In this example, the sequence1 effect specifies a value of `"hide"` for the `filter` property. Therefore, the WipeUp and SetPropertyAction effects only play on those components that change from visible to invisible by setting their `visible` property to `false`.

In the second Sequence effect, you set the `filter` property to `show`. Therefore, the WipeUp and SetPropertyAction effects only play on those components whose state changes from invisible to visible by setting their `visible` property to `true`.

The Move effect also specifies a `filter` property with a value of `move`. Therefore, it applies to all target components that are changing position.

The following table describes the possible values of the `filter` property:

Value	Description
`add`	Specifies to play the effect on all children added during the change of view state.
`hide`	Specifies to play the effect on all children whose `visible` property changes from `true` to `false` during the change of view state.
`move`	Specifies to play the effect on all children whose `x` or `y` properties change during the change of view state.
`remove`	Specifies to play the effect on all children removed during the change of view state.
`resize`	Specifies to play the effect on all children whose `width` or `height` properties change during the change of view state.
`show`	Specifies to play the effect on all children whose `visible` property changes from `false` to `true` during the change of view state.

Example: Using a filter

The following example shows the complete code for the example in "Filtering effects" on page 1065:

```
<?xml version="1.0" ?>
<!-- transitions\FilterShowHide.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"  width="700" >

    <!-- Define two view state, in addition to the base state.-->
    <mx:states>
        <mx:State name="Two">
            <mx:SetProperty target="{p1}" name="visible" value="false"/>
            <mx:SetProperty target="{p2}" name="visible" value="true"/>
            <mx:SetProperty target="{p3}" name="visible" value="true"/>
            <mx:SetProperty target="{p2}" name="x" value="0"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
        </mx:State>
        <mx:State name="Three">
            <mx:SetProperty target="{p1}" name="visible" value="true"/>
            <mx:SetProperty target="{p2}" name="visible" value="false"/>
            <mx:SetProperty target="{p3}" name="visible" value="true"/>
            <mx:SetProperty target="{p1}" name="x" value="0"/>
            <mx:SetProperty target="{p1}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="0"/>
        </mx:State>
    </mx:states>

    <!-- Define a single transition for all state changes.-->
    <mx:transitions>
        <mx:Transition fromState="*" toState="*">
            <mx:Sequence targets="{[p1,p2,p3]}">
                <mx:Sequence id="sequence1" filter="hide" >
                    <mx:WipeUp/>
                    <mx:SetPropertyAction name="visible" value="false"/>
                </mx:Sequence>
                <mx:Move filter="move"/>
                <mx:Sequence id="sequence2" filter="show" >
                    <mx:SetPropertyAction name="visible" value="true"/>
                    <mx:WipeUp/>
                </mx:Sequence>
            </mx:Sequence>
        </mx:Transition>
    </mx:transitions>

    <mx:Canvas id="pm" width="100%" height="100%">
        <mx:Panel id="p1" title="One"
```

```
                x="0" y="0" width="100" height="100"
                click="currentState='' " >
            <mx:Label fontSize="24" text="One"/>
        </mx:Panel>

        <mx:Panel id="p2" title="Two"
                x="0" y="110" width="100" height="100"
                click="currentState='Two'" >
            <mx:Label fontSize="24" text="Two"/>
        </mx:Panel>

        <mx:Panel id="p3" title="Three"
                visible="false"
                width="100" height="100"
                click="currentState='Three'" >
            <mx:Label fontSize="24" text="Three"/>
        </mx:Panel>
    </mx:Canvas>
</mx:Application>
```

Defining a custom filter

Flex lets you use the EffectTargetFilter class to define a custom filter that is executed by each transition effect on each target of the effect. The following table describes the properties of the EffectTargetFilter class:

Property	Description
`filterProperties`	An Array of Strings specifying component properties. If any of the properties in the Array have changed on the target component, play the effect on the target.
`filterStyles`	An Array of Strings specifying style properties. If any of the style properties in the Array have changed on the target component, play the effect on the target.
`filterFunction`	A property containing a reference to a callback function that defines custom filter logic. Flex calls this method on every target of the effect. If the function returns `true`, the effect plays on the target; if it returns `false`, the target is skipped by that effect.

The callback filter function has the following signature:

```
filterFunc(propChanges:Array, instanceTarget:Object):Boolean
{
  // Return true to play the effect on instanceTarget,
  // or false to not play the effect.
}
```

The function takes the following arguments:

`propChanges` An Array of PropertyChanges objects, one object per target component of the effect. If a property of a target is not modified by the transition, it is not included in this Array.

`instanceTarget` The specific target component of the effect that you filter.

For an example using a custom filter function, see "Example: Using a custom effect filter" on page 1075.

To create a filter, you define an EffectTargetFilter object, and then specify that object as the value of the `Effect.customFilter` property for an effect, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initFilter(event);" width="700">

  <mx:Script>
    <![CDATA[

      import mx.effects.EffectTargetFilter;

      // Define the EffectTargetFilter object.
      private var myBlurFilter:EffectTargetFilter;

      // Initialize the EffectTargetFilter object, and set the
      // customFilter property for two Blur effects.
      private function initFilter(event:Event):void {
        myBlurFilter = new EffectTargetFilter();

        // Play the effect on any target that modifies the
        // value of the x or width property.
        myBlurFilter.filterProperties=['x', 'width'];

        myBlur.customFilter=myBlurFilter;
        myUnBlur.customFilter=myBlurFilter;
      }
    ]]>
  </mx:Script>

  <mx:transitions>
    <mx:Transition fromState="*" toState="*">
      <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
        <mx:Blur id="myBlur"/>
          <mx:Parallel>
            <mx:Move  duration="400"/>
            <mx:Resize duration="400"/>
          </mx:Parallel>
        <mx:Blur id="myUnBlur"/>
      </mx:Sequence>
    </mx:Transition>
```

```
  </mx:transitions>
  ...
</mx:Application>
```

You can also add the custom filter in MXML, as the following example shows:

```
<?xml version="1.0" ?>
<!-- transitions\EffectFilterExampleMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="700">

    <mx:states>
        <mx:State name="One">
            <mx:SetProperty target="{p1}" name="x" value="110"/>
            <mx:SetProperty target="{p1}" name="y" value="0"/>
            <mx:SetProperty target="{p1}" name="width" value="500"/>
            <mx:SetProperty target="{p1}" name="height" value="210"/>
            <mx:SetProperty target="{p2}" name="x" value="0"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="100"/>
            <mx:SetProperty target="{p2}" name="height" value="100"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
        <mx:State name="Two">
            <mx:SetProperty target="{p2}" name="x" value="110"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="500"/>
            <mx:SetProperty target="{p2}" name="height" value="210"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
    </mx:states>

    <mx:transitions>
        <mx:Transition fromState="*" toState="*">
            <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
                <mx:Blur id="myBlur" duration="100" blurXFrom="0.0"
                blurXTo="10.0" blurYFrom="0.0" blurYTo="10.0">
                    <mx:customFilter>
                        <mx:EffectTargetFilter
                            filterProperties="['width','x']"/>
                    </mx:customFilter>
                </mx:Blur>
                <mx:Parallel>
                    <mx:Move  duration="400"/>
                    <mx:Resize duration="400"/>
```

```
                </mx:Parallel>
                <mx:Blur id="myUnBlur" duration="100" blurXFrom="10.0"
                blurXTo="0.0" blurYFrom="10.0" blurYTo="0.0">
                    <mx:customFilter>
                        <mx:EffectTargetFilter
                            filterProperties="['width','x']"/>
                    </mx:customFilter>
                </mx:Blur>
            </mx:Sequence>
        </mx:Transition>
    </mx:transitions>

    <mx:Canvas id="pm" width="100%" height="100%">
        <mx:Panel id="p1" title="One"
                x="0" y="0" width="100" height="100"
                click="currentState='One'" >
            <mx:Label fontSize="24" text="One"/>
        </mx:Panel>

        <mx:Panel id="p2" title="Two"
                x="0" y="110" width="100" height="100"
                click="currentState='Two'" >
            <mx:Label fontSize="24" text="Two"/>
        </mx:Panel>

        <mx:Panel id="p3" title="Three"
                x="110" y="0" width="500" height="210"
                click="currentState=''" >
            <mx:Label fontSize="24" text="Three"/>
        </mx:Panel>
    </mx:Canvas>
</mx:Application>
```

Writing a filter function

To create a filter function, you define an EffectTargetFilter object, and then specify that object as the value of the `Effect.customFilter` property for an effect. The following example uses the `creationComplete` event of an application to initialize an EffectTargetFilter object, and then specify it as the value of the `customFilter` property for two Blur effects:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initFilter(event);" width="700">

  <mx:Script>
    <![CDATA[

      import mx.effects.EffectTargetFilter;
      import flash.events.Event;
```

```
        // This function returns true for the Panel moving to x=110.
        public function filterFunc(propChanges:Array,
          instanceTarget:Object):Boolean
        {
          ...
        }

        // Define the EffectTargetFilter object.
        private var myBlurFilter:EffectTargetFilter;

        // Initialize the EffectTargetFilter object, and set the
        // customFilter property for two Blur effects.
        private function initFilter(event:Event):void {
          myBlurFilter = new EffectTargetFilter();

          myBlurFilter.filterFunction=filterFunc;

          myBlur.customFilter=myBlurFilter;
          myUnBlur.customFilter=myBlurFilter;
        }

       ]]>
    </mx:Script>

    <mx:transitions>
      <mx:Transition fromState="*" toState="*">
        <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
          <mx:Blur id="myBlur"/>
            <mx:Parallel>
              <mx:Move  duration="400"/>
              <mx:Resize duration="400"/>
            </mx:Parallel>
          <mx:Blur id="myUnBlur"/>
        </mx:Sequence>
      </mx:Transition>
    </mx:transitions>

    ...

</mx:Application>
```

The `propChanges` argument passed to the filter function contains an Array of PropertyChanges objects, one object per target component of the effect. The following table describes the properties of the PropertyChanges class:

Property	Description
`target`	A target component of the effect. The `end` and `start` properties of the PropertyChanges class define how the target component is modified by the change to the view state.
`start`	An Object that contains the starting properties of the `target` component, as defined by the current view state. For example, for a `target` component that is both moved and resized by a change to the view state, the `start` property contains the starting position and size of the component, as the following example shows: `{x:00, y:00, width:100, height:100}`
`end`	An Object that contains the ending properties of the `target` component, as defined by the destination view state. For example, for a `target` component that is both moved and resized by a change to the view state, the `end` property contains the ending position and size of the component, as the following example shows: `{x:100, y:100, width:200, height:200}`

Within the custom filter function, you first search the `propChanges` Array for the PropertyChanges object that matches the `instanceTarget` argument by comparing the `instanceTarget` argument to the `propChanges.target` property.

The following filter function examines the `propChanges` Array to determine if it should play the effect on the `instanceTarget`. In this example, the filter function returns `true` only for those components being moved to a position where the `x` property equals 110, as the following example shows:

```
// This function returns true for a target moving to x=110.
public function filterFunc(propChanges:Array,
  instanceTarget:Object):Boolean {

  // Determine the length of the propChanges Array.
  for (var i:uint=0; i < propChanges.length; i++)
  {
    // Determine the Array element that matches the effect target.
    if (propChanges[i].target == instanceTarget)
    {
      // Check to see if the end Object contains a value for x.
      if (propChanges[i].end["x"] != undefined)
      {
        // Return true of the end value for x is 110.
        return (propChanges[i].end.x == 110);
      }
    }
```

```
    }
    // Otherwise, return false.
    return false;
}
```

Example: Using a custom effect filter

In the following example, you use a custom filter function to apply Blur effects to one of the three targets of a transition. The other two targets are not modified by the Blur effects.

To determine the target of the Blur effects, the custom filter function examines the `x` property of each target. The Blur effects play only on the component moving to x=110, as the following example shows:

```
<?xml version="1.0" ?>
<!-- transitions\EffectFilterExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initFilter(event);"
    width="700">

    <mx:Script>
        <![CDATA[

            import mx.effects.EffectTargetFilter;
            import flash.events.Event;

            // This function returns true for the Panel moving to x=110.
            public function filterFunc(propChanges:Array,
                    instanceTarget:Object):Boolean {
                // Determine the length of the propChanges Array.
                for (var i:uint=0; i < propChanges.length; i++)
                {
                    // Determine the Array element
                    // that matches the effect target.
                    if (propChanges[i].target == instanceTarget)
                    {
                        // Check whether the end Object contains
                        // a value for x.
                        if (propChanges[i].end["x"] != undefined)
                        {
                            // Return true of the end value for x is 110.
                            return (propChanges[i].end.x == 110);
                        }
                    }
                }

                return false;
            }

            // Define the EffectTargetFilter object.
```

```
        private var myBlurFilter:EffectTargetFilter;

        // Initialize the EffectTargetFilter object, and set the
        // customFilter property for two Blur effects.
        private function initFilter(event:Event):void {
            myBlurFilter = new EffectTargetFilter();

            myBlurFilter.filterFunction=filterFunc;

            myBlur.customFilter=myBlurFilter;
            myUnBlur.customFilter=myBlurFilter;
        }
      ]]>
</mx:Script>

<mx:states>
    <mx:State name="One">
        <mx:SetProperty target="{p1}" name="x" value="110"/>
        <mx:SetProperty target="{p1}" name="y" value="0"/>
        <mx:SetProperty target="{p1}" name="width" value="500"/>
        <mx:SetProperty target="{p1}" name="height" value="210"/>
        <mx:SetProperty target="{p2}" name="x" value="0"/>
        <mx:SetProperty target="{p2}" name="y" value="0"/>
        <mx:SetProperty target="{p2}" name="width" value="100"/>
        <mx:SetProperty target="{p2}" name="height" value="100"/>
        <mx:SetProperty target="{p3}" name="x" value="0"/>
        <mx:SetProperty target="{p3}" name="y" value="110"/>
        <mx:SetProperty target="{p3}" name="width" value="100"/>
        <mx:SetProperty target="{p3}" name="height" value="100"/>
    </mx:State>
    <mx:State name="Two">
        <mx:SetProperty target="{p2}" name="x" value="110"/>
        <mx:SetProperty target="{p2}" name="y" value="0"/>
        <mx:SetProperty target="{p2}" name="width" value="500"/>
        <mx:SetProperty target="{p2}" name="height" value="210"/>
        <mx:SetProperty target="{p3}" name="x" value="0"/>
        <mx:SetProperty target="{p3}" name="y" value="110"/>
        <mx:SetProperty target="{p3}" name="width" value="100"/>
        <mx:SetProperty target="{p3}" name="height" value="100"/>
    </mx:State>
</mx:states>

<mx:transitions>
    <mx:Transition fromState="*" toState="*">
        <mx:Sequence id="t1" targets="{[p1,p2,p3]}">
            <mx:Blur id="myBlur" duration="100" blurXFrom="0.0"
            blurXTo="10.0" blurYFrom="0.0" blurYTo="10.0"/>
            <mx:Parallel>
                <mx:Move  duration="400"/>
                <mx:Resize duration="400"/>
```

```
                </mx:Parallel>
                <mx:Blur id="myUnBlur" duration="100" blurXFrom="10.0"
                blurXTo="0.0" blurYFrom="10.0" blurYTo="0.0"/>
            </mx:Sequence>
        </mx:Transition>
    </mx:transitions>

    <mx:Canvas id="pm" width="100%" height="100%">
        <mx:Panel id="p1" title="One"
                x="0" y="0" width="100" height="100"
                click="currentState='One'" >
            <mx:Label fontSize="24" text="One"/>
        </mx:Panel>

        <mx:Panel id="p2" title="Two"
                x="0" y="110" width="100" height="100"
                click="currentState='Two'" >
            <mx:Label fontSize="24" text="Two"/>
        </mx:Panel>

        <mx:Panel id="p3" title="Three"
                x="110" y="0" width="500" height="210"
                click="currentState=''" >
            <mx:Label fontSize="24" text="Three"/>
        </mx:Panel>
    </mx:Canvas>
</mx:Application>
```

Transition tips and troubleshooting

This section contains some tips and troubleshooting information that you might find helpful when working with transitions.

Tips

Determine the type of transition you are defining:

- With *dynamic* transitions, you know what effects you want to play, but not which targets they will play on.
- With *explicit* transitions, you know exactly what happens to each individual target.

Complex transitions may consist of both dynamic and explicit elements.

Tips for dynamic transitions

List all possible targets in the parent composite effect.

By default, all effects play on all specified targets. Use filtering with dynamic transitions to limit the target set.

Dynamic transitions can be used with a wide variety of state changes.

Tips for explicit transitions

Specify targets on the child effects, or on a composite effect when all child effects of the composite effect have the same targets.

By default, all effects play on all specified targets. For explicit transitions, make sure the targets are correctly set.

Explicit transitions typically require a different transition for each change to the view state.

Troubleshooting

Troubleshooting a transition effect that does not play

Is the effect target being hidden or removed? If so, make sure you add an `<mx:RemoveChild>` property to the view state definition, or an `<mx:SetPropertyAction name="visible">` tag in the transition definition.

Does the change to the view state define settings that pertain to the transition effect? For example, if you have a Resize effect, you must change the `width` or `height` property of the target when the view state changes to trigger the effect.

Check that you specified the correct targets to the transition.

Check that your filter settings on the effect and on any parent effect are not excluding the target.

Troubleshooting an effect playing on too many targets

Add a filter for a dynamic transition, or change the targets for an explicit transition.

Troubleshooting wrong effect parameters

Did you specify explicit parameters on the effect? Are they correct?

Ensure that you correctly set the `showTarget` property for mask effects such as the Iris effect, and the wipe effects.

Troubleshooting a flickering or flashing target

Ensure that you correctly set the `showTarget` property for mask effects such as the Iris effect, and the wipe effects.

Is the effect target being hidden or removed? If so, make sure you add an `<mx:RemoveChild>` property to the view state definition to remove the target, or an `<mx:SetPropertyAction name="visible">` tag in the transition definition to hide the target.

Troubleshooting when an application does not look correct after a transitions

Some effects leave the target with changed properties. For example, a Fade effect leaves the `alpha` property of the target at the `alphaTo` value specified to the effect. If you use the Fade effect on a target that sets `alpha` property to zero, you must set the `alpha` property to a nonzero value before the object appears again.

Try removing one effect at a time from the transition until it no longer leaves your application with the incorrect appearance.

CHAPTER 29

Using the Drag and Drop Manager

29

The Drag and Drop Manager lets you move data from one place in an Adobe Flex application to another. This feature is especially useful in a visual application where your data can be items in a list, images, or Flex components.

This topic describes the drag-and-drop operation, and how to add this functionality to your application.

Contents

About the Drag and Drop Manager

Visual development environments typically let you manipulate objects in an application by selecting them with a mouse and moving them around the screen. The Flex Drag and Drop Manager lets you select an object, such as an item in a List control, or a Flex control, such as an Image control, and then drag it over another component to add it to that component.

All Flex components support the drag-and-drop operation. Flex also includes specific additional support for the drag-and-drop operation for certain controls, such as List, Tree, and DataGrid.

About the drag-and-drop operation

The drag-and-drop operation has three main stages: initiation, dragging, and dropping. The following paragraphs describe these stages from a broad perspective. Later sections describe the stages in greater detail.

Initiation A user initiates a drag-and-drop operation by using the mouse to select a Flex component, or an item in a Flex component, and then moving the component or item while holding down the mouse button. For example, a user selects an item in a List control with the mouse and, while holding down the mouse button, moves the mouse several pixels. The selected component is the *drag initiator*.

Dragging While holding down the mouse button, the user moves the mouse around the Flex application. Flex displays an image during the drag, which is the *drag proxy*. The *drag source* (a DragSource object) contains the data being dragged.

Dropping When the user moves the drag proxy over another Flex component, that component becomes a possible *drop target*. The drop target can inspect the drag source to determine whether the data is in a format that the target accepts and, if so, let the user drop the data onto it. If the drop target determines that the data is not in an acceptable format, the drop target can disallow the drop operation.

Upon a successful drop operation, Flex adds the data to the target and, optionally, deletes it from its original location.

Using drag-and-drop with list-based controls

The following controls include built-in support for the drag-and-drop operation:

- DataGrid
- HorizontalList
- List
- Menu
- PrintDataGrid
- TileList
- Tree

You can make these controls drag initiators by setting the `dragEnabled` property to `true`. You do not have to define an event listener to start the drag operation. Flex lets you move items by dragging them from a drag-enabled control to a drop-enabled control, or copy them by dragging while pressing the Control key.

These controls provide properties and methods for managing the drag-and-drop operation. The following table lists these properties and methods:

Property/ Method	Description
`defaultDrop IndicatorSkin`	Specifies the name of the skin to use for the drop-insert indicator which shows where the dragged data will be inserted. Meaningful only if you call the `showDropFeedback()` method in the target. The default value is `ListDropIndicator`.
`dragEnabled`	A Boolean value that specifies whether the control is a drag initiator. The default value is `false`. When `true`, users can drag selected items from the control. When a user drags items from the control, Flex creates a DragSource object that contains the following data objects: • A copy of the selected item or items in the control. For all controls except for Tree, the format string is `"items"` and the items implement the IDataProvider interface. For Tree controls the format string is `"treeItems"` and the items implement the ITreeDataProvider API interface. • A copy of the initiator, with a format String of `"source"`.
`dropEnabled`	A Boolean value that specifies whether the control can be a drop target that uses default values for handling items dropped onto it. The default value is `false`, which means that you must write event listeners for the drag events. When the value is `true`, you can drop items onto the control by using the default drop behavior.
`dragMoveEnabled`	If the value is `true`, and the `dragEnabled` property is `true`, specifies that you can move or copy items from the drag initiator to the drop target. When you move an item, the item is deleted from the drag initiator when you add it to the drop target. If the value is `false`, you can only copy an item to the drop target. For a copy, the item in the drag initiator is not affected by the drop. When the `dragMoveEnabled` property is `true`, you must hold down the Control key during the drop operation to perform a copy. The default value is `false` for the List and DataGrid controls, and `true` for the Tree control.
`calculateDropIndex`	Returns the item index in the drop target where the item will be dropped. Used by the `dragDrop` event listener to add the items in the correct location. Not available in the TileList or HorizontalList controls.

Property/ Method	Description
`hideDropFeedback()`	Hides drop target feedback and removes the focus rectangle from the target. You typically call this method from within the listener for the `dragExit` and `dragDrop` events.
`showDropFeedback()`	Specifies to display the focus rectangle around the target control and positions the drop indicator where the drop operation should occur. If the control has active scroll bars, hovering the mouse pointer over the control's top or bottom scrolls the contents. You typically call this method from within the listener for the `dragOver` event.

Examples: List control

The following simple example lets you copy items from one List control to another:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToList.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="500" height="200"
    borderStyle="solid"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
            private function initApp():void {
            srclist.dataProvider = ['Reading', 'Television', 'Movies'];
                destlist.dataProvider =[];
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:VBox>
            <mx:Label text="Available Activities"/>
            <mx:List id="srclist"
                height="100"
                allowMultipleSelection="true"
                dragEnabled="true"/>
        </mx:VBox>

        <mx:VBox>
            <mx:Label text="Activities I Like"/>
            <mx:List id="destlist"
                height="100"
                dropEnabled="true"/>
        </mx:VBox>
    </mx:HBox>
</mx:Application>
```

By setting the `dragEnabled` property to `true` on the first List and the `dropEnabled` property to `true` on the second list control, you enabled users to drag items from the first list to the second without worrying about any of the underlying processing.

The default value of the `dragMoveEnabled` property is `false`, so you can only copy elements from one List control to the other. If you modify the example to set `dragMoveEnabled` to `true` in the source List control, you can move and copy elements, as the following example shows:

```
<mx:VBox>
  <mx:Label text="Available Activities"/>
  <mx:List id="srclist"
```

```
    height="100"
    allowMultipleSelection="true"
    dragEnabled="true"
    dragMoveEnabled="true"/>
</mx:VBox>
```

The default action is to move the element. To copy an element, hold down the Control key during the drop operation.

You can allow two-way drag and drop by enabling the `dragEnabled` property and the `dropEnabled` property on both lists as follows:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToListBothWays.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="500" height="200"
    borderStyle="solid"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
            private function initApp():void {
            srclist.dataProvider = ['Reading', 'Television', 'Movies'];
                destlist.dataProvider = [];
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:VBox>
            <mx:Label text="Available Activities"/>
            <mx:List id="srclist"
                height="100"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true"/>
        </mx:VBox>

        <mx:VBox>
            <mx:Label text="Activities I Like"/>
            <mx:List id="destlist"
                height="100"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true"/>
        </mx:VBox>
    </mx:HBox>
</mx:Application>
```

Example: DataGrid control

The following example shows the same principles using the DataGrid control instead of the List control:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleDGToDG.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="700" height="250"
    borderStyle="solid"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
            private function initApp():void {
              srcgrid.dataProvider = [
                {Artist:'Carole King', Album:'Tapestry', Price:11.99},
                {Artist:'Paul Simon', Album:'Graceland', Price:10.99},
                {Artist:'Original Cast', Album:'Camelot', Price:12.99},
               {Artist:'The Beatles', Album:'The White Album', Price:11.99}
              ];

              destgrid.dataProvider =[];
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:VBox>
            <mx:Label text="Available Albums"/>
            <mx:DataGrid id="srcgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true">
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"/>
                    <mx:DataGridColumn dataField="Album"/>
                    <mx:DataGridColumn dataField="Price"/>
                </mx:columns>
            </mx:DataGrid>
        </mx:VBox>

        <mx:VBox>
            <mx:Label text="Buy These Albums"/>
            <mx:DataGrid id="destgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true">
```

```
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"/>
                    <mx:DataGridColumn dataField="Album"/>
                    <mx:DataGridColumn dataField="Price"/>
                </mx:columns>
            </mx:DataGrid>
        </mx:VBox>
    </mx:HBox>
</mx:Application>
```

Example: Tree control

The following example defines a Tree control that lets you drag and drop tree elements within the control to move the elements. If you set `dropEnabled` to `false`, you can drag the elements to other controls, but not move them within the same control.

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleTreeSelf.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="700" height="250"
    borderStyle="solid">

    <mx:Script>
        <![CDATA[
            // Initialize the data provider for the Tree.
            private function initApp():void {
                firstList.dataProvider = treeDP;
            }
        ]]>
    </mx:Script>

    <mx:XML id="treeDP">
        <node label="Mail">
            <node label="Inbox"/>
            <node label="Personal Folder">
                <node label="Demo"/>
                <node label="Personal"/>
                <node label="Saved Mail"/>
                <node label="bar"/>
            </node>
            <node label="Calendar"/>
            <node label="Sent"/>
            <node label="Trash"/>
        </node>
    </mx:XML>

    <mx:Tree id="firstList"
        height="200" width="200"
        showRoot="false"
        labelField="@label"
        dragEnabled="true"
        dropEnabled="true"
        dragMoveEnabled="true"
        allowMultipleSelection="true"
        creationComplete="initApp();"/>
</mx:Application>
```

Notice that the built-in drag-and-drop functionality of the Tree control lets you move items. If you want to copy items, you must implement your own event listeners for the `dragDrop` and `dragComplete` events. For more information on these events, see "Drag-and-drop events" on page 1091.

Manually adding drag-and-drop support

To support drag-and-drop operations with non-list-based controls, or with containers, you must explicitly add support by using a series of special classes and events.

Classes used in drag-and-drop operations

If you don't use the built-in functionality within list classes, you must manually add support for drag-and-drop operations. You use the following classes to implement the drag-and-drop operation:

Class	Function
DragManager	Manages the drag-and-drop operations; for example, its `doDrag()` method starts the drag operation.
DragSource	Identifies and contains the data being dragged. It also provides additional drag management features, such as the ability to add a listener that is called when data is requested.
DragEvent	Represents drag-and-drop events, such as when the user drags over a drag target.

Drag-and-drop events

Flex applications use events to control drag-and-drop operations. A control that acts as a drag initiator can use the following events to manage the drag-and-drop operation:

Event	Description
`mouseDown` and `mouseMove`	The `mouseDown` event is dispatched when the user selects a control with the mouse and holds down the mouse button. The `mouseMove` event is dispatched when the mouse moves. For most controls, you initiate the drag-and-drop operation in response to one of these events. Controls, such as Tree and Grid, that have a `dragEnabled` property, provide built-in support for initiating the drag operation; for these controls you do not use the mouse events.
`dragComplete`	Dispatched when a drag operation completes, either when the drag data drops onto a drop target, or when the drag-and-drop operation ends without performing a drop operation. You can use this event to perform any final cleanup of the drag-and-drop operation. For example, if a user drags a List control item from one list to another, you can use the listener for this event to delete the List control item from the source.

The drop target can use the following events:

Event	Description
`dragEnter`	Dispatched when a drag proxy moves over the target from outside the target. A component *must* define an event listener for this event to be a drop target. In the listener, you can change the appearance of the drop target to provide visual feedback to the user that the component can accept the drag operation. For example, you can draw a border around the drop target, or give focus to the drop target.
`dragOver`	Dispatched while the user moves the mouse over the target, after the `dragEnter` event. You can handle this event to perform additional logic before allowing the drop operation, such as dropping data to various locations within the drop target, reading keyboard input to determine if the drag-and-drop operation is a move or copy of the drag data, or providing different types of visual feedback based on the type of drag-and-drop operation.

Event	Description
`dragDrop`	Dispatched when the user releases the mouse over the target. You can only use this event listener to add the drag data to the drop target.
`dragExit`	Dispatched when the user drags outside the drop target, but does not drop the data onto the target. You can use this event to restore the drop target to its normal appearance if you modified its appearance in response to a `dragEnter` event or other event.

The drag-and-drop operation

The following steps define the drag-and-drop operation:

1. A component becomes a drag-and-drop initiator in either of the following ways:

   Components with a dragEnabled property (These components include List, Tree, DataGrid, PrintDataGrid, Menu, HorizontalList, and TileList.) If the `dragEnabled` property is set to `true`, Flex automatically makes the component an initiator when the user clicks and moves the mouse on the component.

   Components without dragEnabled properties For all other components, the component must sense the user's attempt to start a drag operation and explicitly become an initiator. Typically, you use the `mouseMove` or `mouseDown` event to start the drag-and-drop operation.

   a. The component creates an instance of the mx.core.DragSource class that contains the data to be dragged, and specifies the formats for the data.

   b. The component calls the `mx.managers.DragManager.doDrag()` method, to initiate the drag-and-drop operation.

2. While the mouse button is still pressed, the user moves the mouse around the application. Flex displays the drag proxy image in your application. The `DragManager.defaultDragImageSkin` property defines the default drag proxy image.

NOTE

Releasing the mouse button when the drag proxy is not over a target ends the drag-and-drop operation. Flex generates a `DragComplete` event on the drag initiator, and the `DragManager.getFeedback()` method returns `DragManager.NONE`.

3. If the user moves the drag proxy over a Flex component, Flex dispatches a `dragEnter` event to the component. The component must define an event listener for the `dragEnter` event to be a drop target.

   The `dragEnter` event listener can examine the `DragSource` object to determine whether the data being dragged is in an accepted format. To accept the drop, the event listener calls the `DragManager.acceptDragDrop()` method.

   - If the drop target does not accept the drop, the drop target component's parent chain is examined to determine if any component in the chain accepts the drop data.
   - If the drop target or a parent component accepts the drop, Flex dispatches the `dragOver` event as the user moves the proxy over the target.

4. (Optional) The drag target can handle the `dragOver` event. For example, the target can use this event listener to set the focus on itself.
5. If the user decides not to drop the data onto the drop target and moves the drag proxy outside of the drop target without releasing the mouse button, Flex dispatches a `dragExit` event. The drop target can handle this event; for example, to undo any actions made in the `dragOver` event listener.
6. If the user releases the mouse while over the drop target, Flex dispatches a `dragDrop` event. The drop target's `dragDrop` event listener adds the drag data to the target.
7. If the drop operation completes, Flex dispatches a `dragComplete` event. The drag initiator can handle this event; for example, to delete the data from a data provider.

Example: Accessing the drag data

The following example uses the `dragDrop` event listener to access the data dragged from one DataGrid control to another. This example modifes the example from "Example: DataGrid control" on page 1087 to display the artist's name in an Alert control after a successful drag and drop operation:

```
<?xml version="1.0"?>
<!-- dragdrop\simpleDGToDGAlert.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="700" height="250"
    borderStyle="solid"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
            import mx.events.DragEvent;
            import mx.controls.Alert;

            private function initApp():void {
              srcgrid.dataProvider =  [
                {Artist:'Carole King', Album:'Tapestry', Price:11.99},
                {Artist:'Paul Simon', Album:'Graceland', Price:10.99},
                {Artist:'Original Cast', Album:'Camelot', Price:12.99},
               {Artist:'The Beatles', Album:'The White Album', Price:11.99}
              ];

                destgrid.dataProvider =[];
            }

            // Define the event listener.
            public function dragDropHandler(event:DragEvent):void {
                // Access dragged data as an Array
                // in case multiple items were selected.
                var dragObj:Array=
                    event.dragSource.dataForFormat("items") as Array;

                // Get the Artist for all dragged albums.
                var artistList:String='';
                for (var i:Number = 0; i < dragObj.length; i++) {
                    artistList+='Artist: ' + dragObj[i].Artist + '\n';
                }

                Alert.show(artistList);
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:VBox>
```

```
            <mx:Label text="Available Albums"/>
            <mx:DataGrid id="srcgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true">
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"  />
                    <mx:DataGridColumn dataField="Album" />
                    <mx:DataGridColumn dataField="Price" />
                </mx:columns>
            </mx:DataGrid>
        </mx:VBox>

        <mx:VBox>
            <mx:Label text="Buy These Albums"/>
            <mx:DataGrid id="destgrid"
                allowMultipleSelection="true"
                dragEnabled="true"
                dropEnabled="true"
                dragMoveEnabled="true"
                dragDrop="dragDropHandler(event);">
                <mx:columns>
                    <mx:DataGridColumn dataField="Artist"  />
                    <mx:DataGridColumn dataField="Album" />
                    <mx:DataGridColumn dataField="Price" />
                </mx:columns>
            </mx:DataGrid>
        </mx:VBox>
    </mx:HBox>
</mx:Application>
```

Example: Simple drag-and-drop operation

The following example lets users color a canvas by dropping either of two sample colors onto it. It shows the basic elements of a drag-and-drop operation. The following sections describe these elements in greater detail.

```
<?xml version="1.0"?>
<!-- dragdrop\DandDCanvas.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
    <![CDATA[
        import mx.core.DragSource;
        import mx.managers.DragManager;
        import mx.events.*;
        import mx.containers.Canvas;

        // Called when the user clicks the mouse on either colored canvas.
        // Initializes the drag.
        private function dragIt(event:MouseEvent, text:String,
            format:String):void {

            // Get the drag initiator component from the event object.
            var dragInitiator:Canvas=Canvas(event.currentTarget);
            // Create a DragSource object.
            var ds:DragSource = new DragSource();
            // Add the data to the object.
            ds.addData(text, format);

            // Create a Canvas container to use as the drag proxy.
            // You must specify a size for the proxy image,
            // or else it will not appear.
            var canvasProxy:Canvas = new Canvas();
            canvasProxy.width=30;
            canvasProxy.height=30;
            canvasProxy.setStyle('backgroundColor',
                dragInitiator.getStyle('backgroundColor'));

            // Call the DragManager doDrag() method to start the drag.
            // For information on this method, see
            // the "Initiating the drag" section.
            DragManager.doDrag(dragInitiator, ds, event, canvasProxy);
        }

        // Called if the user dragged a proxy onto the drop target canvas.
        private function doDragEnter(event:DragEvent):void {
            // Get the drop target component from the event object.
            var dropTarget:Canvas=Canvas(event.currentTarget);

            // Accept the drag only if the user is dragging data
```

```
            // identified by the 'color' format value.
            if (event.dragSource.hasFormat('color')) {
                DragManager.acceptDragDrop(dropTarget);
            }
        }

        // Called if the target accepts the dragged object and the user
        // releases the mouse button while over the canvas.
        // Handles the dragDrop event for the List control.
        private function doDragDrop(event:DragEvent):void {
            // Get the data identified by the color format
            // from the drag source.
            var data:Object = event.dragSource.dataForFormat('color');
            // Set the canvas color.
            myCanvas.setStyle("backgroundColor", data);
        }

    ]]>
    </mx:Script>

    <!-- A horizontal box with red and green canvases the user can drag -->
    <mx:HBox>
        <mx:Canvas
            width="30" height="30"
            backgroundColor="red"
            borderStyle="solid"
            mouseMove="dragIt(event, 'red', 'color');"/>
        <mx:Canvas
            width="30" height="30"
            backgroundColor="green"
            borderStyle="solid"
            mouseMove="dragIt(event, 'green', 'color');"/>
    </mx:HBox>

    <mx:Label text="Drag the item into this canvas"/>

    <!-- Handles dragEnter and dragDrop events to allow dropping -->
    <mx:Canvas id="myCanvas"
        width="100" height="100"
        backgroundColor="#FFFFFF"
        borderStyle="solid"
        dragEnter="doDragEnter(event);"
        dragDrop="doDragDrop(event);"/>
</mx:Application>
```

Programming a drag-and-drop operation

The following sections describe how you write an application that uses drag-and-drop operations. It describes how to initiate a drag-and-drop operation, how to specify a drag proxy image, and how to handle the drag-and-drop events.

Starting a drag-and-drop operation

For all controls except the list-based controls, the event listener for the `mouseMove` or `mouseDown` event must explicitly start a drag-and-drop operation. As a general rule, you should use the `mouseMove` event; then, the Drag and Drop Manager only starts the drag-and-drop operation if the mouse is pressed and it is moving. Some applications, such as the one shown in the section "Example: All drag-and-drop events" on page 1106, where the user first selects text using the mouse, can more easily use the `mouseDown` event.

NOTE

For information on starting a drag-and-drop operation for a list-based control, see "Using drag-and-drop with list-based controls" on page 1082.

Writing the initiator event listener

Event listeners that start a drag-and-drop operation often take the following arguments:

Argument	Description
`event`	The event object. The object's `target` property contains a reference to the control that dispatched the event, that is, the control on which the user clicked or moved the mouse.
`data`	An Object that represents the data to drag.
`format`	A String that contains an application-specific identifier for the type of data.

The `format` argument is a text string that you use to label a data format, such as `"color"`, `"list data"`, or `"employee record"`. The drop target can examine this string to determine whether the data format matches the type of data that it accepts. If the format matches, the target lets users drop the data on the target; if the format does not match, the target does not enable the drop operation.

NOTE

The list controls have predefined values for the format. For all controls other than the Tree control, the format String is `"items"`. For the Tree control, the format String is `"treeItems"`. For more information, see "Using drag-and-drop with list-based controls" on page 1082.

The event listener that initiates the drag must do two things:

1. Create a DragSource object and initialize it with the drag data and the format. When the user drops the drag proxy on a control, the control's drag-and-drop events, such as `dragEnter` and `dragDrop`, contain a reference to this object in their `dragSource` property.

   The following code shows the first lines of the event listener function for the `mouseDown` event in "Example: Simple drag-and-drop operation" on page 1096. This code creates the DragSource object and populates it with the data and format:

   ```
   // Called when the user clicks the mouse on either colored canvas.
   // Initializes the drag.
   private function dragIt(event:MouseEvent, text:String,
     format:String):void {

     // Get the drag initiator component from the event object.
     var dragInitiator:Canvas=Canvas(event.currentTarget);
     // Create a DragSource object.
     var ds:DragSource = new DragSource();
     // Add the data to the object.
     ds.addData(text, format);
     ...
   ```

   The event listener first casts the `event.currentTarget` property, which contains the drag initiator, to a Canvas container. This is necessary because the data type of the `event.currentTarget` property is *. After the cast, you can access all of the properties of the Canvas container.

   If you drag large or complex data items, consider creating a listener to copy the data, and specify it by calling the `DragSource.addListener()` method instead of using the `DragSource.addData()` method. If you do this, the data does not get copied until the user drops it, which avoids the processing overhead of copying the data if a user starts dragging data but never drops it. The implementation of the list-based classes use this technique.

2. Call the `DragManager.doDrag()` method to start the drag-and-drop operation.

   The `doDrag()` method has the following signature:

   ```
   doDrag(
     dragInitiator:IUIComponent,
     dragSource:DragSource,
     mouseEvent:MouseEvent,
     dragImage:IFlexDisplayObject = null,
     xOffset:Number = 0, yOffset:Number = 0,
     imageAlpha:Number = 0.5,
     allowMove:Boolean = true):void
   ```

The `doDrag()` method requires three arguments: a reference to the component that initiates the operation (identified by the `event.currentTarget` object); the DragSource object that you created in step 1, and the mouse event that started the drag (the `event` object that was passed to the event listener). Optional arguments specify the drag proxy image and the characteristics of the image.

The following code completes the `mouseDown` event listener function started in step 1 by calling the `doDrag()` method. It creates a Canvas container to use as the drag proxy, and then passes the drag proxy to the DragManager by using the fourth function argument:

```
    // Continued from step 1.
    ...
    // Create a Canvas container to use as the drag proxy.
    // You must specify a size for the proxy image,
    // or else it will not appear.
    var canvasProxy:Canvas = new Canvas();
    canvasProxy.width=30;
    canvasProxy.height=30;
    canvasProxy.setStyle('backgroundColor',
    dragInitiator.getStyle('backgroundColor'));

    // Call the DragManager doDrag() method to start the drag.
    // For information on this method, see
    // the "Initiating the drag" section.
    DragManager.doDrag(dragInitiator, ds, event, canvasProxy);
}
```

For more information on specifying the drag proxy image, see "Specifying the drag proxy image" on page 1101.

Specifying the drag proxy image

You can optionally specify a drag proxy image in the `doDrag()` method of the DragManager class. If you do not specify an image, Flex uses a default proxy image. The `doDrag()` method takes the following optional arguments to specify the image and its properties.

Argument	Description
`dragImage`	A reference to a class or a linkage ID (for an image) that specifies the drag proxy image. To specify a symbol, such as a JPEG image of a product that a user wants to order, use a string that specifies the symbol's name, such as myImage.jpg. To specify a component, such as a Flex container or control, create an instance of the control or container, configure and size it, and then pass it as an argument to the `doDrag()` method.
`xOffset`	A Number that specifies the horizontal offset between the initiator component's upper-left corner and the initial location of the drag proxy image's upper-left corner. You do not normally use this argument for most standard Flex controls.
`yOffset`	A Number that specifies the vertical offset between the initiator component's upper-left corner and the initial location of the drag proxy image's upper-left corner. You do not normally use this argument for most standard Flex controls.
`imageAlpha`	A Number that specifies the alpha value used for the drag proxy image. If omitted, Flex uses an alpha value of 0.5. A value of 0 corresponds to transparent and a value of 1.0 corresponds to fully opaque.

You must specify a size for the drag proxy image, otherwise it does not appear. The following example uses a Canvas object as the drag proxy. You set the background color to the background color of the drag initiator, and set the canvas size to 30 by 30 pixels. The *`imageAlpha`* argument makes the drag proxy opaque.

```
// Create a Canvas container to use as the drag proxy.
// You must specify a size for the proxy image,
// or else it will not appear.
var canvasProxy:Canvas = new Canvas();
canvasProxy.width=30;
canvasProxy.height=30;
canvasProxy.setStyle('backgroundColor',
  dragInitiator.getStyle('backgroundColor'));

// Call the DragManager doDrag() method to start the drag.
DragManager.doDrag(dragInitiator, ds, event, canvasProxy);
```

To use a control with specific contents, such as a VBox control with a picture and label, you must create a custom component that contains the control or controls, and use the component name as the *`dragProxy`* argument.

Handling drag-and-drop events

The following sections describe how to handle events that can happen after a drag-and-drop operation starts. The section for each event includes sample code for the events from an example that lets users drag text selections from one List control to another. The code in "Example: All drag-and-drop events" on page 1106 assembles all the code into a working application.

Handling the dragEnter event

Flex generates a `dragEnter` event when the drag proxy enters any control. A control *must* define a listener for a `dragEnter` event to be a drop target, as follows:

- Within the event listener, you can use the format information in the DragSource object to determine whether the drag data is in a format accepted by the drop target.
- The listener *must* call the `DragManager.acceptDragDrop()` method to enable the user to drop the data on the drop target.
- You can also use this event listener for other actions that you want to occur when the user first drags a drag proxy over a drop target.
- The value of the `action` property of the event object is `DragManager.MOVE`, even if you are doing a copy. This is because the `dragEnter` event occurs before the control recognizes that the Control key is pressed to signal a copy. The `action` property of the event object for the `dragOver` event contains a value that signifies the type of drag operation, and is one of the following: `DragManager.COPY`, `DragManager.LINK`, `DragManager.MOVE`, or `DragManager.NONE`.

In the following `dragEnter` event listener, the event listener writes a message to a TextArea control indicating that the event occurred, and calls the `DragManager.acceptDragDrop()` method to start the drag-and-drop operation:

```
private function doDragEnter(event:DragEvent):void {
  // Get the drop target component from the event object.
  var dragInitiator:List=List(event.currentTarget);

  // Write a message to a TextArea control.
  tiEnter.text += "dragEnter triggered" + "\n";
  DragManager.acceptDragDrop(dragInitiator);
}
```

For a complete example using this event listener, see "Example: All drag-and-drop events" on page 1106.

Handling the dragOver event

The `dragOver` event occurs when the user moves the mouse over a drag-and-drop target whose `dragEnter` event listener has called the `DragManager.acceptDragDrop()` method. Like the `mouseMove` event, this event is dispatched multiple times as the user drags the mouse over the target. The `dragOver` event listener is optional; you do not have to define it to perform a drag-and-drop operation.

The `dragOver` event is useful for specifying the visual feedback that the user gets when the mouse is over a drop target. The `DragEvent` object has three Boolean properties that indicate whether the Control, Alt, and Shift keys are pressed at the time of the event: `ctrlKey`, `altKey`, and `shiftKey`, respectively. You can use these properties to determine whether the user is requesting to move, copy, or link to the dragged item, and can select the appearance of the drag proxy based on the requested operation.

You use the `DragManager.showFeedback()` method to specify the drag-feedback indicator that accompanies the drag proxy. This method uses four constant values for the method argument: `DragManager.COPY`, `DragManager.LINK`, `DragManager.MOVE`, or `DragManager.NONE`. If you specify `DragManager.showFeedBack(DragManager.NONE)`, for example, a red circle with a white *x* appears with the drag proxy. (This is the same image that appears when the user drags over an object that is not a drag target).

You can use the `DragManager.getFeedback()` method in a `dragComplete` event listener to determine the required operations, such as to delete the original copy of the dragged object if the value is `DragManager.MOVE`. For more information, see "Handling the dragDrop event" on page 1104 and "Handling the dragComplete event" on page 1105.

In the following code, the listener determines whether the user is pressing a key while dragging the proxy over the target, and sets the feedback appearance based on the key that is pressed.

```
private function doDragOver(event:DragEvent):void {
  tiOver.text += "dragOver triggered" + "\n";
  if (event.ctrlKey)
    DragManager.showFeedback(DragManager.COPY);
  else if (event.shiftKey)
    DragManager.showFeedback(DragManager.LINK);
  else
    DragManager.showFeedback(DragManager.MOVE);
}
```

For a complete example using this event listener, see "Example: All drag-and-drop events" on page 1106.

Handling the dragDrop event

The `dragDrop` event occurs when the user releases the mouse to drop data on a target. You must define a listener for the event to handle the drop operation and add the data to the target.

```
private function doDragDrop(event:DragEvent):void {
  // Get drop target.
  var dropTarget:List=List(event.currentTarget);
  // Write message to TextArea control.
  tiDrop.text += "dragDrop triggered" + "\n";

  // Hide drop feedback.
  doDragExit(event);

  // Get the dragged items from the drag initiator.
  var items:Array=event.dragSource.dataForFormat("items") as Array;

  // Get the drop location in the destination.
  var dropLoc:int=dropTarget.calculateDropIndex(event);

  // Write status message to TextArea control.
  tiDrop.text += "length: " + String(items.length) + "\n" + "dropLoc: " +
  String(dropLoc) + "\n" + "format: " +
  String(event.dragSource.formats[0])+ "\n";

  // Add each item to the drop target.
  for(var i:uint=0; i < items.length; i++) {
    tiDrop.text += "\n" + "label: " + items[i].label +
      "\n" + String(dropLoc);
    IList(dropTarget.dataProvider).addItemAt(items[i], dropLoc );
  }
}
```

For a complete example using this event listener, see "Example: All drag-and-drop events" on page 1106.

Handling the dragExit event

The `dragExit` event occurs when the user moves the mouse out of a drop target without dropping the data on it. You can define a listener to perform any cleanup on the drop target. In the following example, the listener removes focus box from the target:

```
private function doDragExit(event:DragEvent):void {
  var dropTarget:List=List(event.currentTarget);
  tiExit.text += "dragExit triggered" + "\n";
  dropTarget.hideDropFeedback(event);
}
```

For a complete example using this event listener, see "Example: All drag-and-drop events" on page 1106.

Handling the dragComplete event

The `dragComplete` event occurs after the `dragDrop` event completes. The drag initiator can specify a listener to perform cleanup actions when the drag finishes; or when the target does not accept the drop.

One use of the `dragComplete` event listener is to remove from the drag initiator the objects that you move to the target. The items that you drag from a control are copies of the original items, not the items themselves. Therefore, when you drop items onto the drop target, you use the `dragComplete` event listener to delete them from the drag initiator.

In the following example, the listener writes a message to a TextArea control that indicate that the event occurred:

```
private function doDragComplete(event:DragEvent):void {
  //If the user moves the text, remove it from the source text area.
  if (DragManager.getFeedback() == DragManager.MOVE) {
    TextInput(event.currentTarget).text="";
}
```

For a complete example using this event listener, see "Example: All drag-and-drop events" on page 1106.

Example: All drag-and-drop events

The following example displays each triggered event as you drag items from one List control to another:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDListToListAllEvents.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="500" height="620"
    borderStyle="solid"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
            import mx.events.DragEvent;
            import mx.controls.List;
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.collections.IList;

            private var placeHolder:Object = {label:"First", blah:"blah"};

            private function initApp():void {
                var dp:Array = ([
                    {label:"First", data:"1"},
                    {label:"Second", data:"2"},
                    {label:"Third", data:"3"},
                    {label:"Fourth", data:"4"},
                ]);
                firstList.dataProvider = dp;
                secondList.dataProvider = [];
            }

            private function doDragEnter(event:DragEvent):void {
                var dragInitiator:List=List(event.currentTarget);
                tiEnter.text += "dragEnter triggered" + "\n";
                DragManager.acceptDragDrop(dragInitiator);
            }

            private function doDragOver(event:DragEvent):void {
                tiOver.text += "dragOver triggered" + "\n";
                if (event.ctrlKey)
                    DragManager.showFeedback(DragManager.COPY);
                else if (event.shiftKey)
                    DragManager.showFeedback(DragManager.LINK);
                else
                    DragManager.showFeedback(DragManager.MOVE);
            }

            private function doDragDrop(event:DragEvent):void {
```

```
                // Get drop target.
                var dropTarget:List=List(event.currentTarget);
                // Write message to TextArea control.
                tiDrop.text += "dragDrop triggered" + "\n";

                // Hide drop feedback.
                doDragExit(event);

                // Get the dragged items from the drag initiator.
                var items:Array =
                    event.dragSource.dataForFormat("items") as Array;
                // Get the drop location in the destination.
                var dropLoc:int = dropTarget.calculateDropIndex(event);

                // Write status message to TextArea control.
                tiDrop.text += "length: " + String(items.length) + "\n" +
                    "dropLoc: " + String(dropLoc) + "\n" + "format: " +
                    String(event.dragSource.formats[0])+ "\n";

                // Add each item to the drop target.
                for(var i:uint=0; i < items.length; i++)
                {
                  tiDrop.text += "\n" + "label: " + items[i].label + "\n" +
                    String(dropLoc);
                  IList(dropTarget.dataProvider).addItemAt(items[i],
                    dropLoc);
                }
            }

            private function doDragComplete(event:DragEvent):void {
                tiComplete.text += "dragComplete triggered" + "\n";
            }

            private function doDragExit(event:DragEvent):void {
                var dropTarget:List=List(event.currentTarget);
                tiExit.text += "dragExit triggered" + "\n";
                dropTarget.hideDropFeedback(event);
            }
        ]]>
    </mx:Script>

    <mx:Label text="Drag items from left list to right list to see the
triggered events."/>
    <mx:HBox>
        <mx:List  id="firstList"
            dragEnabled="true"
            allowMultipleSelection="true"
            dragComplete="doDragComplete(event);"
            height="100"/>
```

```
        <mx:List  id="secondList"
            dragEnter="doDragEnter(event);"
            dragExit="doDragExit(event);"
            dragOver="doDragOver(event);"
            dragDrop="doDragDrop(event);"
            dragEnabled="true" height="100"/>
    </mx:HBox>

    <mx:VBox horizontalAlign="right">
        <mx:HBox>
            <mx:Label text="dragEnter: "/>
            <mx:TextArea id="tiEnter" width="200" height="80"/>
        </mx:HBox>

        <mx:HBox>
            <mx:Label text="dragDrop: "/>
            <mx:TextArea id="tiDrop" width="200" height="200"/>
        </mx:HBox>

        <mx:HBox>
            <mx:Label text="dragOver: "/>
            <mx:TextArea id="tiOver" width="200" height="80"/>
        </mx:HBox>

        <mx:HBox>
            <mx:Label text="dragExit: "/>
            <mx:TextArea id="tiExit" width="200" height="80"/>
        </mx:HBox>

        <mx:HBox>
            <mx:Label text="dragComplete: "/>
            <mx:TextArea id="tiComplete" width="200" height="80"/>
        </mx:HBox>
    </mx:VBox>
</mx:Application>
```

Drag-and-drop techniques and considerations

This section documents miscellaneous techniques and considerations that can help you create effective drag-and-drop applications.

Using a container as a drop target

To use a container as a drop target, you must use the `backgroundColor` property of the container to set a color. Otherwise, the background color of the container is transparent, and the Drag and Drop Manager is unable to detect that the mouse pointer is on a possible drop target.

Dragging an image

In the following example, you use the `<mx:Image>` tag to load a draggable image into a Canvas container. In this example, the image is both the drag initiator and the drag proxy. To specify the image as the drag proxy, you specify an Image control as the drag proxy to the `doDrag()` method.

```
<?xml version="1.0"?>
<!-- dragdrop\DandDImageProxy.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            //Import classes so you don't have to use full names.
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;
            import mx.controls.Image;
            import mx.containers.Canvas;

            //Variables used to hold the image's location
            public var xOff:Number;
            public var yOff:Number;

            // Embed icon image.
            [Embed(source='assets/globe.jpg')]
            public var globeImage:Class;

            // Drag initiator event handler, called by
            //  the image's mouseMove event.
            private function dragMe(event:MouseEvent, img1:Image,
                format:String):void {
                var dragInitiator:Image=Image(event.currentTarget);
                var ds:DragSource = new DragSource();
                ds.addData(img1, format);
                // The drag manager uses the image as the drag proxy
                // and sets the alpha to 100 (opaque),
                // so it appears to be dragged across the canvas.
                var imageProxy:Image = new Image();
                imageProxy.source = globeImage;
                imageProxy.height=10;
                imageProxy.width=10;
                DragManager.doDrag(dragInitiator, ds, event,
                    imageProxy, 0, 0, 1.00);
            }

           //Function called by the canvas dragEnter event; enables dropping
            private function doDragEnter(event:DragEvent):void {
                DragManager.acceptDragDrop(Canvas(event.target));
```

```
            }

            // Function called by the canvas dragDrop event;
            // Sets the image object's position,
            // "dropping" it in its new location.
            private function doDragDrop(event:DragEvent,
                target1:Canvas, format:String):void {
                myimg.x = target1.mouseX - xOff
                myimg.y = target1.mouseY - yOff
            }

           // Helper function called by the dragged image's mouseMove event,
            // as the image drags across the canvas.
            // The function updates the xOff and yOff variables to show the
            // current mouse location.
            private function myoffset(img:Image):void {
                xOff = img.mouseX
                yOff = img.mouseY
            }
        ]]>
    </mx:Script>

    <!-- The Canvas is the drag target -->
    <mx:Canvas id="v1"
        width="500" height="500"
        borderStyle="solid"
        backgroundColor="#DDDDDD"
        dragEnter="doDragEnter(event);"
        dragDrop="doDragDrop(event, v1, 'img');">

    <!-- The image is the drag initiator and the drag proxy. -->
        <mx:Image id="myimg"
            source="@Embed(source='assets/globe.jpg')"
            mouseMove="dragMe(event, myimg, 'img'); myoffset(myimg);"/>
    </mx:Canvas>
</mx:Application>
```

CHAPTER 30

Embedding Assets

30

Many Adobe Flex applications use external assets like images, sounds, and fonts. Although you can reference and load assets at run time, you often compile these assets into your applications. The process of compiling an asset into your application is called *embedding the asset*. Flex lets you embed image files, movie files, MP3 files, and TrueType fonts into your applications.

This topic contains an overview of the process that you use to embed an image or a SWF file as an asset into your application. For information on embedding fonts, see Chapter 19, "Using Fonts," on page 763.

Contents

About embedding assets

When you embed an asset, you compile it into your application's SWF file. The advantage of embedding an asset is that it is included in the SWF file, and can be accessed faster than when the application has to load it from a remote location at run time. The disadvantage of embedding an asset is that your SWF file is larger than if you load the asset at run time.

Examples of embedding assets

One of the most common uses of the embed mechanism is to import an image for a Flex control by using the `@Embed()` directive in an MXML tag definition. For example, many controls support icons or skins that you can embed in the application. The Button control lets you specify label text, as well as an optional icon image, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed\ButtonIcon.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Button label="Icon Button" icon="@Embed(source='logo.gif')"/>
</mx:Application>
```

Another option for embedding is to associate the embedded image with a variable by using the `[Embed]` metadata tag. In this way, you can reference the embedded image from multiple locations in your application, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed\ButtonIconClass.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            [Embed(source="logo.gif")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </mx:Script>

    <mx:Button label="Icon Button 1" icon="{imgCls}"/>
    <mx:Button label="Icon Button 2" icon="{imgCls}"/>
</mx:Application>
```

For style properties, you can embed an asset as part of a style sheet definition by using the `Embed()` directive. One common use for style properties is to set a component's skins. For example, you can set skins for a Button control by using the `overSkin`, `upSkin`, and `downSkin` style properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed\ButtonIconCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Style>
        .myCustomButton {
            overSkin:Embed(source="overIconImage.gif");
            upSkin:Embed(source="upIconImage.gif");
            downSkin:Embed(source="downIconImage.gif");
        }
    </mx:Style>

    <mx:Button label="Icon Button Style Def" styleName="myCustomButton"/>
</mx:Application>
```

NOTE The equal sign (=) in the style sheet is a Flex extension that might not be supported by all CSS processors. If you find that it is not supported, you can use the `Embed(filename)` syntax.

Accessing assets at run time

The alternative to embedding an asset is to load the asset at run time. You can load an asset from the local file system in which the SWF file runs, or you can access a remote asset, typically though an HTTP request over a network.

Embedded assets load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded assets also require you to recompile your applications whenever your asset changes.

Assets loaded at run time exist as separate, independent files on your web server (or elsewhere) and are not compiled into your Flex applications. The referenced assets add no additional overhead to an application's initial load time. However, you might experience a delay when you use the asset and load it in Adobe Flash Player. These assets are independent of your Flex application, so you can change them without causing a recompile operation as long as the names of the modified assets remain the same.

For examples that load and asset at run time, see "Image control" on page 325 or "SWFLoader control" on page 319.

For security, by default Flash Player does not allow an application to access some types of remote data at run time, such as SWF files, from a domain other than the domain from which the application was served. Therefore, a server that hosts data must be in the same domain as the server hosting your application, or the server must define a crossdomain.xml file. A crossdomain.xml file is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from specific domains, or from all domains. For more information on application security, see Chapter 4, "Applying Flex Security," in *Building and Deploying Flex 2 Applications*.

Supported file types

You can embed the following types of files in a Flex application:

- GIF files
- JPG and JPEG files
- PNG files
- SVG files
- SWF files
- Symbols stored in a SWF file
- MP3 files
- TTF files
- System fonts

Syntax for embedding assets

The syntax that you use for embedding assets depends on where in your application you embed the asset. Flex supports the following syntaxes:

- The `[Embed(parameter1, paramater2, ...)]` metadata tag

  You use this syntax to embed an asset in an ActionScript file, or within an `<mx:Script>` block in an MXML file. For more information, see "Using the [Embed] metadata tag" on page 1120.
- The `@Embed(parameter1, paramater2, ...)` directive

  You use this syntax in an MXML tag definition to embed an asset. For more information, see "Using the @Embed() directive in MXML" on page 1120.
- The `Embed(parameter1, paramater2, ...)` directive

  You use this syntax within an `<mx:Style>` block in an MXML file to embed an asset. For more information, see "Embedding assets in style sheets" on page 1121.

All three varieties of the embed syntax let you access the same assets; the only difference is where you use them in your application.

Escaping the @ character

You can use the slash character (\ ) to escape the at sign character (@) when you want to use a literal @ character. For example, the string "\@Embed(foo)" means the literal string "@Embed(foo)"). You use two slash characters (\\) to escape a single backslash character. For example, use the character string "\\@" to specify the literal strings "\@".

Embed parameters

Each form of the embed syntax takes one or more optional parameters. The exact syntax that you use to embed assets depends on where they are embedded. Some of these parameters are available regardless of what type of asset you are embedding, and others are specific to a particular type of media. For example, you can use the `source` and `mimeType` parameters with any type of media, but the `scaleGridRight` parameter applies only to images.

The following table describes the parameters available for any type of embedded asset. For more information, see "About the source parameter" on page 1118 and "About the MIME type" on page 1119.

Parameter	Description
`source`	Specifies the name and path of the asset to embed; either an absolute path or a path relative to the file containing the embed statement. The embedded asset must be a locally stored asset. Therefore, you cannot specify a URL for an asset to embed. For more information on setting the path, see "About setting the path to the embedded asset" on page 1118.
`mimeType`	Specifies the mime type of the asset.

The following table describes the parameters specific for images and Sprite objects. For more information, see "Using scale-9 formatting with embedded images" on page 1127.

Parameter	Description
`scaleGridTop`	Specifies the distance, in pixels, of the upper dividing line from the top of the image in a scale-9 formatting system. The distance is relative to the original, unscaled size of the image.
`scaleGridBottom`	Specifies the distance in pixels of the lower dividing line from the top of the image in a scale-9 formatting system. The distance is relative to the original, unscaled size of the image.

Parameter	Description
`scaleGridLeft`	Specifies the distance in pixels of the left dividing line from the left side of the image in a scale-9 formatting system. The distance is relative to the original, unscaled size of the image.
`scaleGridRight`	Specifies the distance in pixels of the right dividing line from the left side of the image in a scale-9 formatting system. The distance is relative to the original, unscaled size of the image.

The following table describes the parameter that is specific to SWF files. For more information, see "Embedding SWF files" on page 1125.

Parameter	Description
`symbol`	Specifies the symbol in a SWF file to embed, for use with Adobe's Macromedia Flash Player 8 and earlier.

About the source parameter

In almost all cases, you must specify the `source` parameter or nothing is embedded.

The `source` parameter is the default parameter of the `[Embed]` metadata tag; therefore, if you are not specifying any other parameters, you can just supply its value without explicitly including the parameter name or assigning it the desired value, as the following example shows:

```
<mx:Style>
  .myCustomButton {
    overSkin:Embed("overIconImage.gif");
    upSkin:Embed(source="upIconImage.gif");
    downSkin:Embed(source="downIconImage.gif");
  }
</mx:Style>
```

About setting the path to the embedded asset

You can specify a fully qualified path to the image or a URL, as the following examples show:

```
<mx:Button label="Icon Button"
  icon="@Embed(source='c:/myapp/assets/logo.gif')"/>
<mx:Button label="Icon Button"
  icon="@Embed(source='http://host.com/myapp/assets/logo.gif')"/>
```

NOTE

Do not use the backslash character, \, as a separator in the path.

If the path does not start with a slash character, Flex first searches for the file relative to the file containing the `[Embed]` metadata tag. For example, the MXML file testEmbed.mxml file includes the following code:

```
<mx:Button label="Icon Button" icon="@Embed(source='assets/logo.gif')"/>
```

In this example, Flex searches the subdirectory named assets in the directory containing the testEmbed.mxml file. If the image is not found, Flex then searches for the image in the SWC files associated with the application.

If the path starts with a slash character, Flex first searches the directory of the MXML file for the asset, and then it searches the source path. You specify the source path to the Flex compiler by using the `source-path` compiler option. For example, you set the `source-path` option as the following code shows:

```
-source-path=a1,a2,a3
```

The MXML file a1/testEmbed.mxml then uses the following code:

```
<mx:Button label="Icon Button" icon="@Embed(source='/assets/logo.gif')"/>
```

Flex first searches for the file in a1/assets, then a2/assets, and then a3/assets. If the image is not found, Flex searches for the image in the SWC files associated with the application.

If the MXML file is in the a2 directory, as in a2/testEmbed.mxml, Flex first searches the a2 directory, and then the directories specified by the `source-path` option.

About the MIME type

You can optionally specify a MIME type for the imported asset by using the `mimeType` parameter. If you do not specify a `mimeType` parameter, Flex makes a best guess about the type of the imported file based on the file extension. If you do specify it, the `mimeType` parameter overrides the default guess of the asset type.

Flex supports the following MIME types:

- application/octet-stream
- application/x-font
- application/x-font-truetype
- application/x-shockwave-flash
- audio/mpeg
- image/gif
- image/jpeg
- image/png
- image/svg

- image/svg-xml

Using the [Embed] metadata tag

You can use the `[Embed]` metadata tag to import JPEG, GIF, PNG, SVG, SWF, TTF, and MP3 files.

You must use the `[Embed]` metadata tag before a variable definition, where the variable is of type Class. The following example loads an image file, assigns it to the `imgCls` variable, and then uses that variable to set the value of the `source` property of an Image control:

```
<?xml version="1.0"?>
<!-- embed\ImageClass.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100" height="80" borderStyle="solid">

    <mx:Script>
        <![CDATA[
            [Embed(source="logo.gif")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </mx:Script>

    <mx:Image source="{imgCls}"/>
</mx:Application>
```

Notice that Flex uses data binding to tie the `imgCls` variable to the `source` property. If you omit the `[Bindable]` metadata tag preceding the `imgCls` variable definition, Flex can only perform the data binding operation once, at application startup. When you include the `[Bindable]` metadata tag, Flex recognizes any changes to the `imgCls` variable, and updates any components that use that variable when a change to it occurs.

Generally, this method of embedding assets provides more flexibility than other methods because you can import an asset once, and then use it in multiple places in your application, and because you can update the asset and have the data binding mechanism propagate that update throughout your application.

Using the @Embed() directive in MXML

Many Flex components, such as Button and TabNavigator, take an `icon` property or other property that lets you specify an image to the control. You can embed the image asset by specifying the property value by using the `@Embed()` directive in MXML. You can use any supported graphic file with the `@Embed()` directive, including SWF files and assets inside SWF files.

You can use the `@Embed()` directive to set an MXML tag property, or to set a property value by using a child tag. The `@Embed()` directive returns a value of type Class or String. If the component's property is of type String, the `@Embed()` directive returns a String. If the component's property is of type Class, the `@Embed()` directive returns a Class. Using the `@Embed()` directive with a property that requires value of any other data type results in an error.

The following example creates a Button control, and sets its `icon` property by using the `@Embed()` directive:

```
<?xml version="1.0"?>
<!-- embed\ButtonAtEmbed.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Button label="Icon Button" icon="@Embed(source='logo.gif')"/>

</mx:Application>
```

Embedding assets in style sheets

Many style properties of Flex components support imported assets. Most frequently, you use these style properties to set the skins for a component. *Skinning* is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of images, SWF files, or class files that contain drawing API methods.

For more information on skinning, and embedding asset using style sheets, see Chapter 20, "Using Skins," on page 805.

Embedding asset types

This section describes how to import various types of media, including images, SWF files, and sound files.

Embedding JPEG, GIF, and PNG images

Flex supports embedding JPEG, GIF, and PNG files. You can use these images for icons, skins, and other types of application assets.

You might want to manipulate an embedded image at run time. To manipulate it, you determine the data type of the object representing the image, and then use the appropriate ActionScript methods and properties of the object.

For example, you use the [Embed] metadata tag in ActionScript to embed a GIF image, as the following code shows:

```
[Embed(source="logo.gif")]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded image. When embedding JPEG, GIF, and PNG files, Flex defines imgCls as a reference to a subclass of the mx.core.BitmapAsset class, which is a subclass of the flash.display.BitmapData class.

In ActionScript, you can create and manipulate an object that represents the embedded image before passing it to a control. To do so, you create an object with the type of the embedded class, manipulate it, and then pass the object to the control, as the following example shows:

```
<?xml version="1.0"?>
<!-- embed/EmbedAccessClassObject.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[

            import mx.core.BitmapAsset;

            [Embed(source="logo.gif")]
            [Bindable]
            public var imgCls:Class;

            public var varOne:String = "This is a public variable";

            private function modImage():void {
                // Create an object from the embed class.
                // Since the embedded image is a GIF file,
                // the data type is BitmapAsset.
                var imgObj:BitmapAsset = new imgCls() as BitmapAsset;

                // Modify the object.
                imgObj.bitmapData.noise(4);

                // Write the modified object to the Image control.
                myImage.source=imgObj;
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:Image id="myImageRaw" source="{imgCls}"/>
        <mx:Image id="myImage" creationComplete="modImage();"/>
        <mx:Text id="myText" text="foo"/>
    </mx:HBox>
</mx:Application>
```

In this example, the first Image control displays the unaltered image and the second Image control displays the modified image. You use the `bitmapData` property of the mx.core.BitmapAsset class to modify the object. The `bitmapData` property is of type flash.display.BitmapData; therefore, you can use all of the methods and properties of the BitmapData class to manipulate the object.

Embedding SVG images

Flex supports importing Scalable Vector Graphics (SVG) images, or a GZip compressed SVG image in a SVGZ file, into an application. This lets you import SVG images and use SVG images as icons for Flex controls.

Flex supports a subset of the SVG 1.1 specification to let you import static, two-dimensional scalable vector graphics. This includes support for basic SVG document structure, Cascading Style Sheets (CSS) styling, transformations, paths, basic shapes, and colors, and a subset of text, painting, gradients, and fonts. Flex does not support SVG animation, scripting, or interactivity with the imported SVG image.

For example, you use the `[Embed]` metadata tag in ActionScript to embed an SVG image, as the following code shows:

```
[Embed(source="logo.svg")]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded image. When embedding an SVG image, Flex defines imgCls as a reference to a subclass of the mx.core.SpriteAsset class, which is a subclass of the flash.display.Sprite class. Therefore, you can manipulate the image by using the methods and properties of the SpriteAsset class. For an example that manipulates an imported image, see "Embedding JPEG, GIF, and PNG images" on page 1121.

Embedding sounds

Flex supports embedding MP3 sound files for later playback. The following example creates a simple media player with Play and Stop buttons:

```
<?xml version="1.0"?>
<!-- embed/EmbedSound.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import flash.media.*;

            [Embed(source="sample.mp3")]
            [Bindable]
            public var sndCls:Class;

            public var snd:Sound = new sndCls() as Sound;
            public var sndChannel:SoundChannel;

            public function playSound():void {
                sndChannel=snd.play();
            }

            public function stopSound():void {
                sndChannel.stop();
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:Button label="play" click="playSound();"/>
        <mx:Button label="stop" click="stopSound();"/>
    </mx:HBox>
</mx:Application>
```

In this example, you define a class named sndCls that represents the embedded MP3 file. When embedding MP3 files, Flex defines sndCls as a reference to a subclass of the mx.core.SoundAsset, which is a subclass of the flash.media.Sound class.

Flex can handle any legal filename, including filenames that contain spaces and punctuation marks. If the MP3 filename includes regular quotation marks, ensure that you use single quotation marks around the filename.

You do not have to embed the sound file to use it with Flex. You can also use the Sound class to load a sound file at run time. For more information, see the Sound class in the *Adobe Flex 2 Language Reference*.

Embedding SWF files

Flex fully supports embedding Flash SWF files. This section describes how to embed different types of SWF files.

Embedding SWF files for Flash Player 8 and earlier

You can embed SWF files created for Flash Player 8 and earlier. When embedded, your Flex 2 application cannot interact with the embedded SWF file. That is, you cannot use ActionScript in a Flex 2 application to access the properties or methods of a SWF file created for Flash Player 8 or earlier.

NOTE	You can use the flash.net.LocalConnection class to communicate between a Flex 2 application and a SWF file created for Flash Player 8 or earlier.

For example, you use the `[Embed]` metadata tag in ActionScript to embed a SWF file, as the following code shows:

```
[Embed(source="icon.swf")]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded SWF file. Flex defines imgCls as a reference to a subclass of the mx.core.MovieClipLoaderAsset class, which is a subclass of the flash.disaply.MovieClip class. Therefore, you can manipulate the image by using the methods and properties of the MovieClipLoaderAsset class.

Embedding SWF symbols

Flex lets you reference exported symbols in an embedded SWF file. If the symbol has dependencies, Flex embeds them also; otherwise, Flex embeds only the specified symbol from the SWF file. To reference a symbol, you specify the `symbol` parameter, as follows:

```
[Embed(source='SWFFileName.swf', symbol='symbolName')]
```

NOTE	Flash defines three types of symbols: Button, MovieClip, and Graphic. You can embed Button and MovieClip symbols in a Flex application, but you cannot embed a Graphic symbol because it cannot be exported for ActionScript.

This capability is useful when you have a SWF file that contains multiple exported symbols, but you want to load only some of them into your Flex application. Loading only the symbols required by your application makes your resulting Flex SWF file smaller than if you imported the entire SWF file.

A Flex application can import any number of SWF files. However, if two SWF files have the same filename and the exported symbol names are the same, you cannot reference the duplicate symbols, even if the SWF files are in separate directories.

If the SWF file contains any ActionScript code, Flex prints a warning during compilation, and then stripped out the ActionScript from the embed symbol. This means that you can only embed the symbol itself.

The following example imports a green square from a SWF file that contains a library of different shapes:

```
<?xml version="1.0"?>
<!-- embed\EmbedSWFSymbol.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:VBox id="vBox0" width="300" height="300" >
        <mx:Image id="image0"
           source="@Embed(source='circleSquare.swf', symbol='greenSquare')"
        />
    </mx:VBox>
</mx:Application>
```

If you use the `[Embed]` metadata tag in ActionScript to embed the symbol, you can access the object that represents the symbol, as the following code shows:

```
[Embed(source='shapes.swf', symbol='greenSquare')]
[Bindable]
public var imgCls:Class;
```

In this example, you define a class named imgCls that represents the embedded symbol. Internally, Flex defines imgCls as a reference to a subclass of either one of the following classes:

SpriteAsset For single-frame SWF files.

MovieClipLoaderAsset For multiframe SWF files.

Embedding SWF files that represent Flex 2 applications

You can embed SWF files that represent Flex 2 applications. For example, you use the `[Embed]` metadata tag in ActionScript to embed a SWF file, as the following code shows:

```
[Embed(source="flex2.swf")]
[Bindable]
public var flexAppCls:Class;
```

In this example, you define a class named flexAppCls that represents the embedded SWF file. Flex defines flexAppCls as a reference to a subclass of the mx.core.MovieClipLoaderAsset class, which is a subclass of the flash.disaply.MovieClip class. Therefore, you can manipulate the embedded SWF file by using the methods and properties of the MovieClipLoaderAsset class.

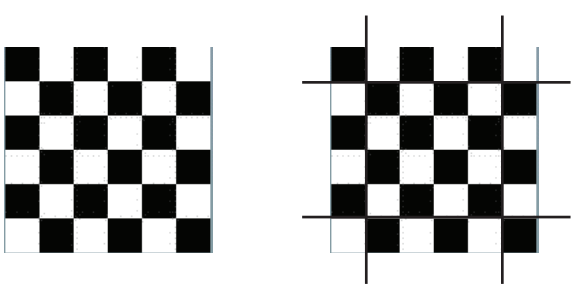
You typically embed a Flex 2 application when you do not require the embedding application to interact with the embedded application. If the embedded application requires interactivity with the embedded application, you might consider implementing it as a custom component, rather than as a separate application.

Alternatively, if you use the SWFLoader control to load the Flex 2 application at run time, the embedding application can interact with the loaded application to access its properties and methods. For more information and examples, see "Interacting with a loaded Flex 2 application" on page 321.

Using scale-9 formatting with embedded images

Flex supports scale-9 formatting of embedded images. The scale-9 formatting feature lets you define nine sections of an image that scale independently. The nine regions are defined by two horizontal lines and two vertical lines running through the image, which form the inside edges of a 3 by 3 grid. For images with borders or fancy corners, scale-9 formatting provides more flexibility than full-graphic scaling.

The following example show an image, and the same image with the regions defined by the scale-9 borders:



When you scale an embedded image that uses scale-9 formatting, all text and gradients are scaled normally. However, for other types of objects the following rules apply:

- Content in the center region is scaled normally.
- Content in the corners is not scaled.
- Content in the top and bottom regions is scaled only horizontally. Content in the left and right regions is scaled only vertically.

- All fills (including bitmaps, video, and gradients) are stretched to fit their shapes.

If you rotate the image, all subsequent scaling is normal, as if you did not define any scale-9 formatting.

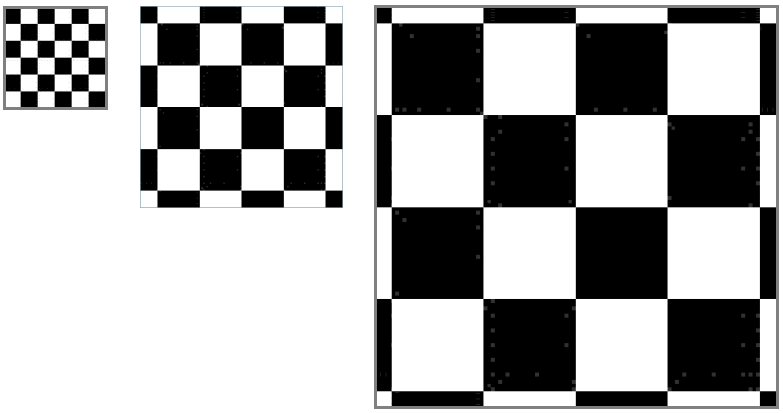
To use scale-9 formatting, define the following four parameters within your embed statement: `scaleGridTop`, `scaleGridBottom`, `scaleGridLeft`, and `scaleGridRight`. For more information on these parameters, see "Embed parameters" on page 1117.

An embedded SWF file may already contain scale-9 information specified by using Flash Professional. In that case, the SWF file ignores any scale-9 parameters that you specify in the embed statement.

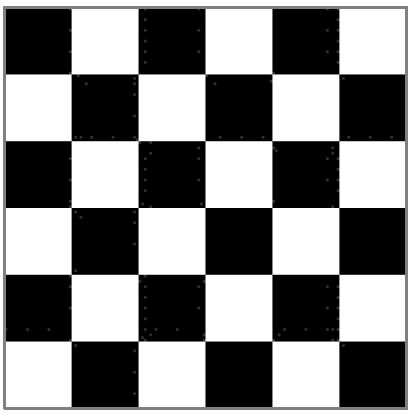
The following example creates a Flex logo with border that uses scale-9 formatting to maintain a set border, regardless of how the image itself is resized:

```
<?xml version="1.0"?>
<!-- embed\Embed9slice.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="1200" height="600">

    <mx:Script>
        <![CDATA[
            [Embed(source="slice_9_grid.gif",
                scaleGridTop="25", scaleGridBottom="125",
                scaleGridLeft="25", scaleGridRight="125")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:Image source="{imgCls}"/>
        <mx:Image source="{imgCls}" width="300" height="300"/>
        <mx:Image source="{imgCls}" width="450" height="450"/>
    </mx:HBox>
</mx:Application>
```

The original image is 30 by 30 pixels. The preceding code produces a resizable image that maintains a 5-pixel border:



If you had omitted the scale-9 formatting, the scaled image would have appeared exactly like the unscaled image, as the following image shows:



In this example, you define a class named imgCls that represents the embedded image. If the image is a SWF file that uses scale-9 formatting, Flex defines imgCls as a subclass of the mx.core.SpriteAsset class, which is a subclass of the flash.display.Sprite class. Therefore, you can use all of the methods and properties of the SpriteAsset class to manipulate the object.

CHAPTER 31

Creating Modular Applications

31

You can create modules that you dynamically load in your Flex applications.

Contents

Modular applications overview

This section describes modules and how they are used by modular applications.

About modules

Modules are SWF files that can be loaded and unloaded by an application. They cannot be run independently of an application, but any number of applications can share the modules.

Modules let you split your application into several pieces, or modules. The main application, or shell, can dynamically load other modules that it requires, when it needs them. It does not have to load all modules when it starts, nor does it have to load any modules if the user does not interact with them. When the application no longer needs a module, it can unload the module to free up memory and resources.

Modular applications have the following benefits:

- Smaller initial download size of the SWF file.
- Shorter load time due to smaller SWF file size.
- Better encapsulation of related aspects of an application. For example, a “reporting” feature can be separated into a module that you can then work on independently.

Benefits of modules

Modules are similar to Runtime Shared Libraries (RSLs) in that they separate code from an application into separately loaded SWF files. Modules are much more flexible than RSLs because modules can be loaded and unloaded at run time and compiled without the application.

Two common scenarios in which using modules is beneficial are a large application with different user paths and a portal application.

An example of the first common scenario is an enormous insurance application that includes thousands of screens, for life insurance, car insurance, health insurance, dental insurance, travel insurance, and veterinary pet insurance.

Using a traditional approach to rich Internet application (RIA) design, you might build a monolithic application with a hierarchical tree of MXML classes. Memory use and start-up time for the application would be significant, and the SWF file size would grow with each new set of functionality.

When using this application, however, any user accesses only a subset of the screens. By refactoring the screens into small groups of modules that are loaded on demand, you can improve the perceived performance of the main application and reduce the memory use. Also, when the application is separated into modules, developers' productivity may increase due to better encapsulation of design. When rebuilding the application, the developers also have to recompile only the single module instead of the entire application.

An example of the second common scenario is a system with a main portal application, written in ActionScript 3, that provides services for numerous portlets. Portlets are configured based on data that is downloaded on a per-user basis. Using the traditional approach, you might build an application that compiles in all known portlets. This is inefficient, both for deployment and development.

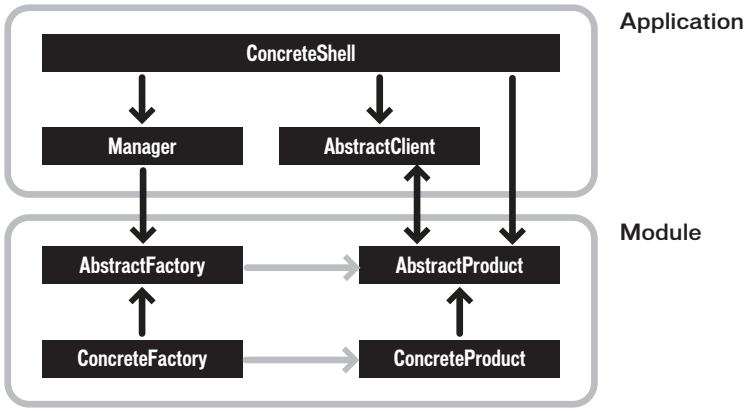
By using modules, you can establish an interface that contains portal services, and a generic portlet interface. You can use XML data to determine which modules to load for a given session. When the module is loaded, you obtain a handle to a class factory inside the module, and from that you create an instance of a class that implements the portlet interface. In this scenario, full recompilation is necessary only if the interfaces change.

Module API details

Modules implement a class factory with a standard interface. The product of that class factory implements an interface known to the shell, or the shell implements an interface known to the modules.

By using shared interface definitions, these shared interfaces reduce hard dependencies between the shell and the module. This provides type-safe communication and enforces an abstraction layer without adding significantly to the SWF file size.

The following image shows the relationship between the shell and the module's interfaces:



Creating modular applications

To create a modular application, you create separate classes for each module, plus an application that loads the modules.

To create a modular application:

1. Create any number of modules. An MXML-based module file's root tag is `<mx:Module>`. ActionScript-based modules extend the ModuleBase class.
2. Compile each module as if it were an application. You can do this by using the mxmlc command-line compiler or the compiler built into Adobe Flex Builder.
3. Create an Application class. This is typically an MXML file whose root tag is `<mx:Application>`, but it can also be an ActionScript-only application.
4. In the Application file, use an `<mx:ModuleLoader>` tag to load each of the modules. You can also do this by using the `load()` method of the mx.modules.ModuleLoader class. For classes that extend ModuleBase, you should use the methods of the ModuleManager class to load them.

The following sections describes these steps in detail.

Creating modules

Modules are classes just like application files. To create a module in ActionScript, you create a file that extends the mx.modules.ModuleBase class. To create a module in MXML, you extend the mx.modules.Module class by creating a file whose root tag is `<mx:Module>`. In that tag, ensure that you add any namespaces that are used in that module. You should also include a type declaration tag at the beginning of the file.

The following example is a module that includes a Chart control:

```
<?xml version="1.0"?>
<!-- modules/ColumnChartModule.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%" >
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            {Month:"Jan", Profit:2000, Expenses:1500},
            {Month:"Feb", Profit:1000, Expenses:200},
            {Month:"Mar", Profit:1500, Expenses:500}
        ]);
    ]]></mx:Script>
    <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
</mx:Module>
```

MXML-based modules can load other modules. Those modules can load other modules, and so on.

Compiling modules

You compile the module as you would compile any Flex application using the mxmlc command-line compiler or the Flex Builder compiler. The following command is the simplest mxmlc command:

```
mxmlc MyModule.mxml
```

The result is a SWF file that you load into your application as a module. You cannot run the module-based SWF file as a stand-alone Flash application or load it into a browser window. It must be loaded by an application as a module.

Controlling module size

Module size varies based on the components and classes that are used in the module. By default, a module includes all framework code that its components depend on, which can cause modules to be large by linking classes that overlap with the application's classes.

To reduce the size of the modules, you can instruct the module to externalize classes that are included by the application. This includes custom classes and framework classes. The result is that the module only includes the classes it requires, while the framework code and other dependencies are included in the application.

To externalize framework classes, you generate a linker report from the application that loads the modules, by using mxmlc commands. You then use this report as input to the module's `load-externs` compiler option.

To create a linker report:

1. Generate the linker report:

   ```
   mxmlc -link-report=report.xml MyApplication.mxml
   ```

2. Compile the application SWF file:

   ```
   mxmlc MyApplication.mxml
   ```

3. Compile the module:

   ```
   mxmlc -load-externs=report.xml MyModule.mxml
   ```

Recompiling modules

You must recompile the modules if you make changes. Recompiling the main application does not trigger a recompilation of the modules. Similarly, if you change the application file, you do not have to recompile the modules, unless you make changes that might affect the linker report or common code.

NOTE

If you externalize the module's dependencies by using the `load-externs` option, your module might not be compatible with future versions of Adobe Flex. You might be required to recompile the module. To ensure that a future Flex application can use a module, compile that module with all the classes it requires. This also applies to applications that you load inside other applications.

Loading and unloading modules

To load and unload modules you use the `load()` and `unload()` methods of the ModuleLoader class. These methods take no parameters; the ModuleLoader loads or unloads the module that matches the value of the current `url` property.

The following example loads and unloads the module when you click the button:

```
<?xml version="1.0"?>
<!-- modules/ASModuleLoaderApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
        import mx.modules.*;

        public function createModule(m:ModuleLoader, s:String):void {
            if (!m.url) {
                m.url = s;
                return;
            }
            m.loadModule();
        }

        public function removeModule(m:ModuleLoader):void {
            m.unloadModule();
        }
        ]]>
    </mx:Script>

    <mx:Panel title="Module Example"
        height="90%"
        width="90%"
        paddingTop="10"
        paddingLeft="10"
```

```
        paddingRight="10"
        paddingBottom="10"
    >
        <mx:TabNavigator id="tn"
            width="100%"
            height="100%"
            creationPolicy="auto"
        >
            <mx:VBox id="vb1" label="Column Chart Module">
                <mx:Button
                    label="Load"
                    click="createModule(chartModuleLoader, l1.text)"
                />
                <mx:Button
                    label="Unload"
                    click="removeModule(chartModuleLoader)"
                />
                <mx:Label id="l1" text="ColumnChartModule.swf"/>
               <mx:ModuleLoader id="chartModuleLoader"/>
            </mx:VBox>

            <mx:VBox id="vb2" label="Form Module">
                <mx:Button
                    label="Load"
                    click="createModule(formModuleLoader, l2.text)"
                />
                <mx:Button
                    label="Unload"
                    click="removeModule(formModuleLoader)"
                />
                <mx:Label id="l2" text="FormModule.swf"/>
                <mx:ModuleLoader id="formModuleLoader"/>
            </mx:VBox>
        </mx:TabNavigator>
    </mx:Panel>
</mx:Application>
```

Setting the location of a ModuleLoader triggers a call to the `loadModule()` method, too. This occurs when you first create a ModuleLoader with the `url` property set. It also occurs if you change the value of that property.

The following example loads the modules without calling the `loadModule()` method because the `url` property is set on the `<mx:ModuleLoader>` tags:

```
<?xml version="1.0"?>
<!-- modules/URLModuleLoaderApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Panel
        title="Module Example"
        height="90%"
        width="90%"
        paddingTop="10"
        paddingLeft="10"
        paddingRight="10"
        paddingBottom="10"
    >

        <mx:Label width="100%" color="blue"
            text="Select the tabs to change the panel."/>

        <mx:TabNavigator id="tn"
            width="100%"
            height="100%"
            creationPolicy="auto"
        >
            <mx:VBox id="vb1" label="Column Chart Module">
                <mx:Label id="l1" text="ColumnChartModule.swf"/>
                <mx:ModuleLoader url="ColumnChartModule.swf"/>
            </mx:VBox>

            <mx:VBox id="vb2" label="Form Module">
                <mx:Label id="l2" text="FormModule.swf"/>
                <mx:ModuleLoader url="FormModule.swf"/>
            </mx:VBox>

        </mx:TabNavigator>
    </mx:Panel>
</mx:Application>
```

When you load a module, Flex ensures that there is only one copy of a module loaded, no matter how many times you call the `load()` method for that module.

Modules are loaded into the child of the current application domain. You can specify a different application domain by using the `applicationDomain` property of the ModuleLoader class.

When two classes of the same name but different implementations are loaded, the first one loaded is the one that is used.

Loading modules from different servers

To load a module from one server into an application running on a different server, you must establish a trust between the module and the application that loads it.

To allow access across domains:

1. In your loading application, you must call the `allowDomain()` method and specify the target domain from which you load a module. So, specify the target domain in the preinitialize event handler of your application to ensure that the application is set up before the module is loaded.
2. In the cross-domain file of the remote server where your module is, add an entry that specifies the server on which the loading application is running.
3. Load the cross-domain file on the remote server in the preinitialize event handler of your loading application.
4. In the loaded module, call the `allowDomain()` method so that it can communicate with the loader.

The following example shows the `init()` method of the loading application:

```
public function setup():void {
  Security.allowDomain("remoteservername");
  Security.loadPolicyFile("http://remoteservername/crossdomain.xml");
  var request:URLRequest = new URLRequest("http://remoteservername
    /crossdomain.xml");
  var loader:URLLoader = new URLLoader();
  loader.load(request);
}
```

The following example shows the loaded module's `init()` method:

```
public function initMod():void {
  Security.allowDomain("loaderservername");
}
```

The following example shows the cross-domain file that resides on the remote server:

```
<!-- crossdomain.xml file located at the root of the server -->
<cross-domain-policy>
  <allow-access-from domain="loaderservername" to-ports="*"/>
</cross-domain-policy>
```

For more information about using the cross-domain policy file, see Chapter 4, "Applying Flex Security," in *Building and Deploying Flex 2 Applications*.

Using ModuleLoader events

The ModuleLoader class triggers several events, including `setup`, `ready`, `loading`, `unload`, `progress`, `error`, and `urlChanged`. You can use these events to track the progress of the loading process, and find out when a module has been unloaded or when the ModuleLoader's target URL has changed.

Using the error event

The `error` event gives you an opportunity to gracefully fail when a module does not load for some reason. In the following example, you can load and unload a module by using the Button controls. To trigger an `error` event, change the URL in the TextInput control to a module that does not exist. The error handler displays a message to the user and writes the error message to the trace log.

```
<?xml version="1.0"?>
<!-- modules/ErrorEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
        import mx.events.ModuleEvent;
        import mx.modules.*;
        import mx.controls.Alert;

        private function errorHandler(e:ModuleEvent):void {
            Alert.show("There was an error loading the module." +
                " Please contact the Help Desk.");
            trace(e.errorText);
        }

        public function createModule():void {
            chartModuleLoader.url = ti1.text;
            chartModuleLoader.loadModule();
        }

        public function removeModule():void {
            chartModuleLoader.unloadModule();
        }

        ]]>
    </mx:Script>

    <mx:Panel title="Module Example"
        height="90%"
        width="90%"
        paddingTop="10"
        paddingLeft="10"
```

```
        paddingRight="10"
        paddingBottom="10"
    >
        <mx:HBox>
            <mx:Label text="URL:"/>
           <mx:TextInput width="200" id="ti1" text="ColumnChartModule.swf"/
>
            <mx:Button label="Load" click="createModule()"/>
            <mx:Button label="Unload" click="removeModule()"/>
        </mx:HBox>
       <mx:ModuleLoader id="chartModuleLoader" error="errorHandler(event)"/
>
    </mx:Panel>
</mx:Application>
```

Using the progress event

You can use the `progress` event to track the progress of a module as it loads. When you add a listener for the `progress` event, Flex calls that listener at regular intervals during the module's loading process. Each time the listener is called, you can look at the `bytesLoaded` property of the event. You can compare this to the `bytesTotal` property to get a percentage of completion.

The following example reports the level of completion during the module's loading process. It also produces a simple progress bar that shows users how close the loading is to being complete.

```
<?xml version="1.0"?>
<!-- modules/SimpleProgressEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
        import mx.events.ModuleEvent;
        import flash.events.ProgressEvent;
        import mx.modules.*;

        [Bindable]
        public var progBar:String = "";
        [Bindable]
        public var progMessage:String = "";

        private function progressEventHandler(e:ProgressEvent):void {
            progBar += ".";
            progMessage =
                "Module " +
                Math.round((e.bytesLoaded/e.bytesTotal) * 100) +
                "% loaded";
        }

        public function createModule():void {
            chartModuleLoader.loadModule();
        }

        public function removeModule():void {
            chartModuleLoader.unloadModule();
            progBar = "";
            progMessage = "";
        }
        ]]>
    </mx:Script>

    <mx:Panel title="Module Example"
        height="90%"
        width="90%"
```

```
        paddingTop="10"
        paddingLeft="10"
        paddingRight="10"
        paddingBottom="10"
    >
        <mx:HBox>
            <mx:Label id="l2" text="{progMessage}"/>
            <mx:Label id="l1" text="{progBar}"/>
        </mx:HBox>

        <mx:Button label="Load" click="createModule()"/>
        <mx:Button label="Unload" click="removeModule()"/>

        <mx:ModuleLoader
            id="chartModuleLoader"
            url="ColumnChartModule.swf"
            progress="progressEventHandler(event)"
        />
    </mx:Panel>
</mx:Application>
```

You can also connect a module loader to a ProgressBar control. The following example creates a custom component for the ModuleLoader that includes a ProgressBar control. The ProgressBar control displays the progress of the module loading.

```
<?xml version="1.0"?>
<!-- modules/MySimpleModuleLoader.mxml -->
<mx:ModuleLoader xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            private function clickHandler():void {
                if (!url) {
                    url="ColumnChartModule.swf";
                }
                loadModule();
            }
        ]]>
    </mx:Script>

    <mx:ProgressBar
        id="progress"
        width="100%"
        source="{this}"
    />
    <mx:HBox width="100%">
      <mx:Button
        id="load"
        label="Load"
        click="clickHandler()"
      />
      <mx:Button
        id="unload"
        label="Unload"
        click="unloadModule()"
      />
      <mx:Button
        id="reload"
        label="Reload"
        click="unloadModule();loadModule();"
      />
    </mx:HBox>
</mx:ModuleLoader>
```

You can use this module in a simple application, as the following example shows:

```
<?xml version="1.0"?>
<!-- modules/ComplexProgressEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:local="*">

    <mx:Panel title="Module Example"
        height="90%"
        width="90%"
        paddingTop="10"
        paddingLeft="10"
        paddingRight="10"
        paddingBottom="10"
    >
        <mx:Label text="Use the buttons below to load and unload
            the module."/>
        <local:MySimpleModuleLoader id="customLoader"/>
    </mx:Panel>

</mx:Application>
```

This example does not change the ProgressBar's `label` property for all events. For example, if you load and then unload the module, the `label` property remains at "LOADING 100%". To adjust the label properly, you must define other event handlers for the ModuleLoader events, such as `unload` and `error`.

CHAPTER 32

Using the History Manager

32

The Adobe Flex History Manager lets users navigate through a Flex application by using the web browser's back and forward navigation commands.

Contents

About history management

The Flex History Manager lets users navigate through a Flex application by using the web browser's back and forward navigation commands. For example, a user can navigate through several Accordion container panes in a Flex application, and then click the browser's Back button to return the application to its previous states.

History management is implemented as a set of files that are referenced in the application's wrapper. By default, Adobe Flex Builder generates a wrapper that supports history management. The web-tier compiler included in Flex Data Services also generates a wrapper that supports history management by default. In both cases, you can disable history management support.

If you are using the command-line compiler to create your Flex application, you must create your own wrapper or use one of the wrapper templates. You can customize your wrapper to include support for history management. For more information, see "Using history management in a custom wrapper" on page 1157. For information on writing a custom wrapper, see Chapter 16, "Creating a Wrapper," in *Building and Deploying Flex 2 Applications*.

For Flex components, history management is automatically supported by navigator containers such as the Accordion and TabNavigator containers. You can also use the HistoryManager class in ActionScript to provide custom history management for other objects in an application, and to call the HistoryManager class's methods.

Using standard history management

History management is enabled by default for the Accordion and TabNavigator navigator containers. This means that if the user selects one of the panes in an Accordion control, that user can return to the previous pane by using the browser's Back button or back navigation command. History management is disabled by default for the ViewStack navigator container.

When history management is enabled, as the user navigates within different navigator containers in an application, each navigation state is saved. Selecting the web browser's Back or Forward button displays the previous or next navigation state that was saved. History management keeps track of where you are in an application, but it is not an undo and redo feature that remembers what you have done.

NOTE	When history management is enabled for a particular component, such as a navigator container, only the state of the navigator container is saved. The state of any of the navigator container's child components is not saved unless history management is specifically added for that component.

For information about how the navigation state is saved and restored, see "How the HistoryManager class saves and loads states" on page 1156.

You can disable or enable history management for a navigator container by setting the container's `historyManagementEnabled` property to `false` or `true`, respectively. The following example shows a TabNavigator container with history management disabled:

```
<mx:TabNavigator historyManagementEnabled="false">
```

In the following example, the user's panel selections are saved for the first Accordion container because it uses default settings, but the second Accordion container has the `historyManagementEnabled` property explicitly set to `false`. When the user selects the web browser's back or forward command, the previous or next state is displayed for the first container, but not for the second.

```
<?xml version="1.0"?>
<!-- historymanager/DisableHistoryManagement.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="800">

  <!-- History management is enabled by default for this Accordion. -->
  <mx:Accordion width="100%" height="50%">
     <mx:VBox label="History management is ENABLED">
        <mx:TextInput text="View 1"/>
     </mx:VBox>
     <mx:VBox label="View 2">
        <mx:TextInput text="View 2"/>
     </mx:VBox>
  </mx:Accordion>

  <!-- History management is disabled for this Accordion. -->
  <mx:Accordion historyManagementEnabled="false" width="100%" height="50%">
     <mx:VBox label="History management is DISABLED.">
        <mx:TextInput text="View 1"/>
     </mx:VBox>
     <mx:VBox label="View 2">
        <mx:TextInput text="View 2"/>
     </mx:VBox>
  </mx:Accordion>
</mx:Application>
```

You can disable history management for an entire application by removing history management support from the wrapper. For more information, see "Using history management in a custom wrapper" on page 1157.

Using custom history management

You can make any custom component history management-aware by performing the following operations on that component:

1. Implement the mx.managers.IHistoryManagerClient interface.
2. Register the component with the HistoryManager's `register()` method.
3. Save the component's state when its state has changed.
4. Implement the `saveState()` and `loadState()` methods of the IHistoryManagerClient interface. These methods have the following signatures:

```
public function saveState():Object
public function loadState(state:Object):void;
```

These steps are described in more detail in the following sections.

Implementing the IHistoryManagerClient interface

To implement the IHistoryManagerClient interface, you can use the `implements` tag attribute; for example:

```
<mx:CheckBox
  xmlns:mx="http://www.adobe.com/2006/mxml"
  implements="mx.managers.IHistoryManagerClient"
  ...
>
```

Registering components with the HistoryManager

To register a component with the HistoryManager class, you call the HistoryManager class's `register()` method with a reference to a component instance that implements the IHistoryManagerClient interface; for example:

```
<mx:CheckBox
    xmlns:mx="http://www.adobe.com/2006/mxml"
    implements="mx.managers.IHistoryManagerClient"
    creationComplete="mx.managers.HistoryManager.register(this);"
>
```

You typically do this when the component has finished initialization.

Saving the component's state

You save the state of the component when its state has changed so that the history manager has a state that it can return to. You typically do this in an event handler by calling the static `HistoryManager.save()` method.

The following example calls the `boxChanged()` method in the change event handler:

```
<mx:CheckBox
  xmlns:mx="http://www.adobe.com/2006/mxml"
  implements="mx.managers.IHistoryManagerClient"
  creationComplete="mx.managers.HistoryManager.register(this);"
  change="boxChanged(event)">
<mx:Script><![CDATA[
  import mx.managers.HistoryManager;
  ...

  // Saves the box's current state.
  private function boxChanged(e:Event):void {
    HistoryManager.save();
  }
]]></mx:Script>
```

Implementing the loadState() and saveState() methods

To use the HistoryManager class for a registered component, you must implement the IHistoryManagerClient interface and include `saveState()` and `loadState()` methods to save and load the state information you want. The `saveState()` method returns an object that contains property:value pairs that represent the current navigation state of a component.

The HistoryManager class contains a `load()` method that calls the `loadState()` method for each registered component with an object identical to the one that the `saveState()` method returns.

Components that implement IHistoryManagerClient also must implement the `loadState()` method; components that extend UIComponent automatically inherit the `loadState()` method.

The following example implements the `loadState()` and `saveState()` methods for a custom CheckBox control:

```
<?xml version="1.0"?>
<!-- historymanager/MyCheckBox.mxml -->
<mx:CheckBox
     xmlns:mx="http://www.adobe.com/2006/mxml"
     label="Check me"
     selected="false"
     implements="mx.managers.IHistoryManagerClient"
     creationComplete="mx.managers.HistoryManager.register(this);"
     change="boxChanged(event)">

  <mx:Script><![CDATA[
        import mx.managers.HistoryManager;

        // Returns an object that contains property:value pairs that
        // represent the current navigation state of a component.
        public function saveState():Object {
           return {selected:selected};
        }

        // Sets the selected property, depending on the state.
        public function loadState(state:Object):void {
           var newState:Boolean = state;

           if (newState != selected) {
            selected = newState;
           } else {
            if (newState) {
                selected = false;
            } else {
                selected = true;
            }
           }
        }

        // Saves the box's current state.
        private function boxChanged(e:Event):void {
           HistoryManager.save();
        }
  ]]></mx:Script>

</mx:CheckBox>
```

The application that uses this control might look like the following (if the MXML files are in the same directory):

```
<?xml version="1.0"?>
<!-- historymanager/CheckBoxApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:local="*">

  <local:MyCheckBox/>

</mx:Application>
```

When the user checks and unchecks the custom CheckBox control, they can use the browser's forward and back buttons to return to the previous state of the control.

An application's total navigation state is limited to the maximum URL size supported by the user's web browser, so you should write the `saveState()` method for a component to save the least amount of data possible. For example, you can write a `saveState()` method for a List control that saves just the `selectedIndex` property.

Using history management with a list-based control control

When using history management with a list-based control, you typically save the selected index of the control, and use that index to determine the state.

The following example is an MXML component, HistoryList.mxml, that registers with the HistoryManager and implements the `saveState()` and `loadState()` methods. The component lets the user browse through a List control.

```
<?xml version="1.0"?>
<!-- historymanager/HistoryList.mxml -->
<mx:List xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initList()"
  change="listChanged()"
  implements="mx.managers.IHistoryManagerClient"
>
  <mx:Script><![CDATA[
     import mx.managers.HistoryManager;

     // Register with the HistoryManager.
     private function initList():void {
        HistoryManager.register(this);
     }

     // Saves the application's current state.
     private function listChanged():void {
        HistoryManager.save()
     }

     // Save the List index.
     public function saveState():Object {
        return { selectedIndex: selectedIndex };
     }

     // Load the List index.
     public function loadState(state:Object):void {
        var newIndex:int = state ? int(state.selectedIndex) : -1;
        if (newIndex != selectedIndex)
        selectedIndex = newIndex;
     }
  ]]></mx:Script>
</mx:List>
```

The following example shows an application file that uses the HistoryList.mxml component:

```
<?xml version="1.0"?>
<!-- historymanager/HistoryListApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:local="*" width="400" height="400" verticalGap="20">

<mx:Script>
<![CDATA[

[Bindable]
private var listData:Array = ["Flex", "Dreamweaver",
"Flash", "Breeze", "Contribute"];
]]>
</mx:Script>

<local:HistoryList id="list1" dataProvider="{listData}"
width="120" height="120"/>

<mx:TextInput id="text1"
text="{list1.selectedItem? list1.selectedItem : ''}"/>
</mx:Application>
```

In this example, you populate the HistoryList control with an Array of Strings. As you select an item in the HistoryList control, it also appears in the TextInput control. Use the browser's back button to cycle back through your selections.

Calling the HistoryManager class's static methods

When you use a ViewStack, Accordion, or TabNavigator container, the HistoryManager's `save()` method is invoked automatically when the user navigates through an application. When you register a component with the HistoryManager class, you must explicitly call the HistoryManager `save()` method to save the state of the component.

The `save()` method and the `register()` and `unregister()` methods, which let you register and unregister a component, are static methods that you can call from your ActionScript code. The following table describes these methods:

Method	Description
`register(`*`component`*`)`	Registers a component with the HistoryManager class; for example: `HistoryManager.register(myList);`

Method	Description
`save()`	Saves the current navigation state of all components registered with the HistoryManager class; for example: `HistoryManager.save();`
`unregister(component)`	Unregisters a component from the HistoryManager class; for example: `HistoryManager.unregister(myList);`

How the HistoryManager class saves and loads states

The history management feature uses the `navigateToURL()` global function to load an invisible HTML frame in the current web browser window. It encodes a Flex application's navigation states into the invisible frame's URL query parameters. A SWF file, called history.swf, in the invisible frame decodes the query parameters and sends the navigation state back to the HistoryManager class. This section describes how the HistoryManager class encodes navigation data into a URL query string and then decodes that navigation data to restore navigation states.

Encoding navigation state data

The HistoryManager class's `save()` method collects the state object that the `saveState()` method returns for each registered component. The `save()` method encodes each property of each object into a query string that uses the standard `prop1=value1&prop2=value2` format. The state ID of the appropriate registered component is prepended with a dash to each property name to identify which component each property belongs to.

A state ID is a four-character hex string, such as `3da7`, that is a hashed form of the full pathname of the registered component within the application's visual hierarchy.

For example, for a TabNavigator container with a state ID of `3da7` that returns the following state object:

```
{selectedIndex:5}
```

the query string is:

```
3da7-selectedIndex=5
```

Decoding and restoring navigation state data

The history.swf file passes stored state properties to a Flex application in a single object. The HistoryManager class extracts the state IDs from the properties in this object and rebuilds state objects for each registered component. The state objects are constructed from the Application object down through its children. For example, when an Accordion container is the third item in a ViewStack container, the ViewStack container must be set to its third item before the Accordion container's navigation state is restored.

Using history management in a custom wrapper

When you place a Flex application inside a custom wrapper instead of generating the wrapper automatically, you must set up the wrapper to support history management if you want to use it. You can configure Flex Builder to generate wrappers that support history management. The web-tier compiler included in Adobe Flex Data Services can also generate a wrapper that supports history management.

In addition, you can view wrapper templates that support history management in the following directories in the /resources/html-templates directory:

- /client-side-detection-with-history
- /express-installation-with-history
- /no-player-detection-with-history

For more information about customizing the wrapper, see Chapter 16, "Creating a Wrapper," in *Building and Deploying Flex 2 Applications*.

Using history management without Flex Data Services

To support history management without Flex Data Services, you must deploy the following files with your application:

- history.js
- history.htm
- history.swf

These files are located in any of the /resources/html-templates/*directory_name*-with-history directories. You should not have to modify these files to use history management, but you must edit your application's wrapper to reference these files.

The following steps are required to support history management without Flex Data Services:

1. Open your wrapper in a text editor.
2. Import the history.js file, as the following example shows:

   ```
   <script language="JavaScript" type="text/JavaScript" src="history.js"/>
   ```

3. Insert the history.htm file in an iframe element at the bottom of the page, as the following example shows:

   ```
   <iframe name="_history.htm" src="history.htm" width="22" height="0"/>
   ```

   You must set the `name` property of the iframe to _history.htm (with the underscore). You can set the `height` property to any value you want, but the `width` property must be a minimum of 22 in order to work in Internet Explorer. Users do not interact with the history.htm page, so it does not need to be visible on the page.
4. Add the `historyUrl` and `lconid` parameters to the `flashVars` variable for both the `object` and `embed` tags, as the following example shows.

   You must add these parameters in JavaScript, because history management uses a JavaScript variable called lc_id.

   ```
   document.write("flashvars='historyUrl=history%5Fhtml&lconid=" +
     lc_id'");
   ```

5. Deploy the history.js and history.html files to your web server and ensure that their locations match those set in the wrapper.
6. Deploy the history.swf file to your web server and ensure that it is in the same location as the history.htm file.

   If you want to change the location of the history.swf file, be sure to update the path to the history.swf file in the history.htm file.

If your wrapper supports history management but does not include any player version detection logic or support for Express Install, you must combine the contents of the history.js and myscript.js files.

Using history management with Flex Data Services

When you are using Flex Data Services, you can call special Flex internal actions that generate the files to support history management. The default configuration of Flex Data Services adds these special actions to the wrapper. This wrapper is generated when a client requests an MXML file only. In general, you must modify your custom wrapper to add the history management support.

The following steps are required to support history management with Flex Data Services:

1. Include the following text at the top of the HTML document, using the correct context root for your web application:

   ```
   <script language='javascript' charset='utf-8'
     src='/flex/flex-internal?action=history_js'></script>
   ```

2. Add the `historyUrl` and `lconid` parameters to the `flashVars` variable for both the `object` and `embed` tags, as the following example shows.

   You must add these parameters in JavaScript, because history management uses a JavaScript variable called lc_id.

   ```
   document.write("
     flashvars='historyUrl=%2Fflex%2Fflex%2Dinternal%3Faction%3Dhistory%5F
     html&lconid=" + lc_id'");
   ```

3. Add the `_history` iframe element, as the following example shows (note that the value of the `width` property must be at least 22, although the `height` property can be any value):

   ```
   <iframe src='/flex/flex-internal?action=history_html' name='_history'
     frameborder='0' scrolling='no' width='22' height='0'></iframe>
   ```

CHAPTER 33

Printing

33

Many Adobe Flex applications let users print from within the application. For example, you might have an application that returns confirmation information after a user completes a purchase. Your application can allow users to print the information on the page to keep for their records.

This topic describes the options for printing when you use the mx.printing.FlexPrintJob and mx.printing.PrintDataGrid classes.

NOTE	You can also print by using the context menu in Adobe Flash Player 9, or the Flash ActionScript PrintJob class, which is documented in *ActionScript 3.0 Language Reference*.

Contents

About printing by using Flex classes

The Flex mx.printing package contains three classes that facilitate the creation of printing output from Flex applications:

FlexPrintJob A class that prints one or more objects. Automatically splits large objects for printing on multiple pages and scales the output to fit the page size.

PrintDataGrid A subclass of the DataGrid control with a default appearance that is customized for printing. The class includes properties and a method that provide additional sizing and printing features.

FlexPrintJobScaleType Defines constants used in the FlexPrintJob `addObject()` method.

Together, these classes give you control over how the user prints information from the application. For example, your application can print only a selected subset of the information on the screen, or it can print information that is not being displayed. Also, your application can reformat the information and optimize its layout and appearance for printing.

Users can print to PostScript and non-PostScript printers, including the Adobe PDF and Macromedia® FlashPaper™ from Adobe® printer drivers.

Using the FlexPrintJob class

You use the FlexPrintJob class to print one or more Flex objects, such as a Form or VBox container. For each object that you specify, Flex prints the object and all objects that it contains. The objects can be all or part of the displayed interface, or they can be components that format data specifically for printing. The FlexPrintJob class lets you scale the output to fit the page, and automatically uses multiple pages to print an object that does not fit on a single page.

You can use the FlexPrintJob class to print a dynamically rendered document that you format specifically for printing. This capability is especially useful for rendering and printing such information as receipts, itineraries, and other displays that contain external dynamic content, such as database content and dynamic text.

You often use the FlexPrintJob class within an event listener. For example, you can use a Button control with an event listener that prints some or all of the application.

NOTE	The FlexPrintJob class causes the operating system to display a Print dialog box. You cannot print without some user action.

Building and sending a print job

You print output by building and sending a print job, as outlined in the following procedure.

To build and send a print job:

1. Create an instance of the FlexPrintJob class:

   ```
   var printJob:FlexPrintJob = new FlexPrintJob();
   ```

2. Start the print job:

   ```
   printJob.start();
   ```

   This causes the operating system to display a Print dialog box.

3. Add one or more objects to the print job and specify how to scale them:

   ```
   printJob.addObject(myObject, FlexPrintJobScaleType.MATCH_WIDTH);
   ```

   Each object starts on a new page.

4. Send the print job to the printer:

   ```
   printJob.send();
   ```

5. Free up any unneeded objects.

NOTE	Because you are spooling a print job to the user's operating system between your calls to the `start()` and `send()` methods, you should limit the code between these calls to print-specific activities. For example, the Flash content should not interact with the user between the `start()` and `send()` methods.

The following sections detail the procedures you use in these steps.

Starting a print job

To start a print job, you create an instance of the FlexPrintJob class and call its `start()` method. This method prompts Flash Player to spool the print job to the user's operating system, which causes the user's operating system to display a Print dialog box.

If the user selects an option to begin printing from the Print dialog box, the `start()` method returns a value of `true`. If the user cancels the print job, the return value is `false`. After the user exits the operating system Print dialog box, the `start()` method uses the printer information to set values for the FlexPrintJob object's `pageHeight` and `pageWidth` properties, which represent the dimensions of the printed page area.

NOTE	Depending on the user's operating system, an additional dialog box might appear until spooling is complete and the application calls the `send()` method.

Only one print job can be active at a time. You cannot start a second print job until one of the following has happened with the previous print job:

- The `start()` method returns a value of `false` (the job failed).
- The `send()` method completes execution following a successful call to the `addObject()` method. Because the `send()` method is synchronous, code that follows it can assume that the call completed successfully.

Adding objects to the print job

You use the `addObject()` method of the FlexPrintJob class to add objects to the print job. Each object starts on a new page; therefore, the following code prints a DataGrid control and a Button control on separate pages:

```
printJob.addObject(myDataGrid);
printJob.addObject(myButton);
```

Scaling a print job

The *`scaleType`* parameter of the `addObject()` method determines how to scale the output. Use the following FlexPrintJobScaleType class constants to specify the scaling method:

Constant	Action
MATCH_WIDTH	(Default) Scales the object to fill the available page width. If the resulting object height exceeds the page height, the output spans multiple pages.
MATCH_HEIGHT	Scales the object to fill the available page height. If the resulting object width exceeds the page width, the output spans multiple pages.
SHOW_ALL	Scales the object to fit on a single page, filling one dimension; that is, it selects the smaller of the MATCH_WIDTH or MATCH_HEIGHT scale types.
FILL_PAGE	Scales the object to fill at least one page completely; that is, it selects the larger of the MATCH_WIDTH or MATCH_HEIGHT scale types.
NONE	Does not scale the output. The printed page has the same dimensions as the object on the screen. If the object height, width, or both dimensions exceed the page width or height, the output spans multiple pages.

If an object requires multiple pages, the output splits at the page boundaries. This can result in unreadable text or inappropriately split graphics. For information on how to format your print job to avoid these problems, see "Printing multipage output" on page 1171.

The FlexPrintJob class includes two properties that can help your application determine how to scale the print job. These properties are read-only and are initially 0. When the application calls the `start()` method and the user selects the Print option in the operating system Print dialog box, Flash Player retrieves the print settings from the operating system. The `start()` method populates the following properties:

Property	Type	Unit	Description
`pageHeight`	Number	Points	Height of the printable area on the page; does not include any user-set margins.
`pageWidth`	Number	Points	Width of the printable area on the page; does not include any user-set margins.

NOTE

A *point* is a print unit of measurement that is 1/72 of an inch. Flex automatically maps 72 pixels to one inch (72 points) of printed output, based on the printer settings.

Completing the print operation

To send the print job to a printer after using the FlexPrintJob `addObject()` method, use the `send()` method, which causes Flash Player to stop spooling the print job so that the printer starts printing.

After sending the print job to a printer, if you use print-only components to format output for printing, call `removeChild()` to remove the print-specific component. For more information, see "Using a print-specific output format" on page 1167.

Example: A simple print job

The following example prints a DataGrid object exactly as it appears on the screen, without scaling:

```
<?xml version="1.0"?>
<!-- printing\DGPrint.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.printing.*;

            // Create a PrintJob instance.
            private function doPrint():void {
                // Create an instance of the FlexPrintJob class.
                var printJob:FlexPrintJob = new FlexPrintJob();

                // Start the print job.
                if (printJob.start() != true) return;

                // Add the object to print. Do not scale it.
                printJob.addObject(myDataGrid, FlexPrintJobScaleType.NONE);

                // Send the job to the printer.
                printJob.send();
            }
        ]]>
    </mx:Script>

    <mx:VBox id="myVBox">
        <mx:DataGrid id="myDataGrid" width="300">
            <mx:dataProvider>
                <mx:Object Product="Flash" Code="1000"/>
                <mx:Object Product="Flex" Code="2000"/>
                <mx:Object Product="ColdFusion" Code="3000"/>
                <mx:Object Product="JRun" Code="4000"/>
            </mx:dataProvider>
        </mx:DataGrid>
        <mx:Button id="myButton"
            label="Print"
            click="doPrint();"/>
    </mx:VBox>
</mx:Application>
```

In this example, selecting the Button control invokes the doPrint() event listener. The event listener creates an instance of the FlexPrintJob class to print the DataGrid control, adds the DataGrid control to the print job using the `addObject()` method, and then uses the `send()` method to print the page.

To print the DataGrid and Button controls on your page, specify myVBox, the ID of the object that contains both controls, in the `addObject()` method's *`object`* parameter. If want to print the DataGrid and Button controls on separate pages, specify each object in a separate `addObject()` method.

To print the DataGrid control so that it spans the page width, omit the second `addObject()` method parameter, or specify `FlexPrintJobScaleType.MATCH_WIDTH`. To print the DataGrid control with the largest size that fits on a single page, specify `FlexPrintJobScaleType.SHOW_ALL`. In the previous example, `FlexPrintJobScaleType.SHOW_ALL` has the same result as `FlexPrintJobScaleType.MATCH_WIDTH` because the DataGrid is short.

Using a print-specific output format

In most cases, you do not want your printed output to look like the screen display. The screen might use a horizontally oriented layout that is not appropriate for paper printing. You might have display elements on the screen that would be distracting in a printout. You might also have other reasons for wanting the printed and displayed content to differ; for example, you might want to omit a password field.

To print your output with print-specific contents and appearance, use separate objects for the screen layout and the print layout, and share the data used in the screen layout with the print layout. You then use the print layout in your call or calls to the FlexPrintJob `addObject()` method.

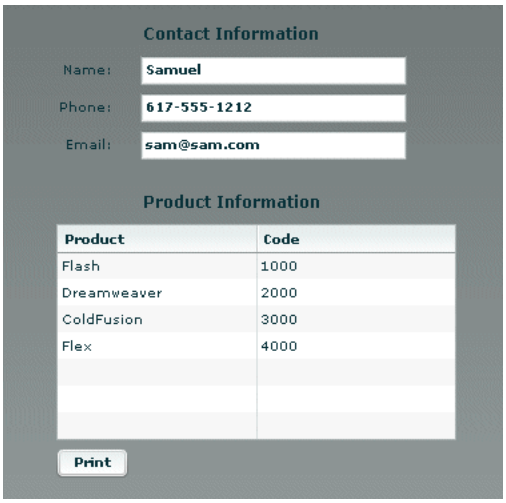
If your form includes a DataGrid control, use the PrintDataGrid control in your print layout. The PrintDataGrid control has two advantages over the DataGrid control for printed output:

- It has a default appearance that is designed specifically for printing.
- It has properties and methods that support printing grids that contain multiple pages of data.

The code in "Example: A simple print-specific output format" on page 1168 uses a short PrintDataGrid control. For more information on using the PrintDataGrid control, see "Using the PrintDataGrid control for multipage grids" on page 1172.

Example: A simple print-specific output format

The example in this section creates a Flex form with three text boxes for entering contact information, and a data grid that displays several lines of product information. The following image shows how the output of this application looks on the screen:



The following image shows how the output looks when the user clicks the Print button to print the data:

Contact: Samuel 617-555-1212 sam@sam.com

Product	Code
Flash	1000
Dreamweaver	2000
ColdFusion	3000
Flex	4000

In this example, the MXML application file displays the screen and controls the printing. A separate custom MXML component defines the appearance of the printed output.

When the user clicks the Print button, the application's `doPrint()` method does the following things:

1. Creates and starts the print job to display the operating system's Print dialog box.
2. After the user starts the print operation in the Print dialog box, creates a child control using the myPrintView component.
3. Sets the MyPrintView control's data from the form data.

4. Sends the print job to the printer.

5. Cleans up memory by removing the print-specific child component.

The printed output does not include the labels from the screen, and the application combines the text from the screen's three input boxes into a single string for printing in a Label control.

The following code shows the contents of the application file:

```
<?xml version="1.0"?>
<!-- printing\DGPrintCustomComp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="450"
    width="550">

    <mx:Script>
        <![CDATA[
            import mx.printing.FlexPrintJob;
            import myComponents.MyPrintView;

            public function doPrint():void {
                // Create a FlexPrintJob instance.
                var printJob:FlexPrintJob = new FlexPrintJob();

                // Start the print job.
                if(printJob.start()) {
                    // Create a MyPrintView control as a child
                    // of the current view.
                    var formPrintView:MyPrintView = new MyPrintView();
                    addChild(formPrintView);

                    // Populate the print control's contact label
                    // with the text from the form's name,
                    // phone, and e-mail controls.
                    formPrintView.contact.text =
                        "Contact: " + custName.text + "  " +
                        custPhone.text + "  " + custEmail.text;

                   // Set the print control's data grid data provider to be
                    // the displayed data grid's data provider.
                    formPrintView.myDataGrid.dataProvider =
                        myDataGrid.dataProvider;

                    // Add the SimplePrintview control to the print job.
                    // For comparison, try setting the
                    // second parameter to "none".
                    printJob.addObject(formPrintView);

                    // Send the job to the printer.
                    printJob.send();
```

```
                    // Remove the print-specific control to free memory.
                    removeChild(formPrintView);
                }
            }
        ]]>
    </mx:Script>

    <!-- The form to display-->
    <mx:Form id="myForm">
        <mx:FormHeading label="Contact Information"/>
        <mx:FormItem label="Name: ">
            <mx:TextInput id="custName"
                width="200"
                text="Samuel Smith"
                fontWeight="bold"/>
        </mx:FormItem>
        <mx:FormItem label="Phone: ">
            <mx:TextInput id="custPhone"
                width="200"
                text="617-555-1212"
                fontWeight="bold"/>
        </mx:FormItem>
        <mx:FormItem label="Email: ">
            <mx:TextInput id="custEmail"
                width="200"
                text="sam@sam.com"
                fontWeight="bold"/>
        </mx:FormItem>

        <mx:FormHeading label="Product Information"/>
        <mx:DataGrid id="myDataGrid" width="300">
            <mx:dataProvider>
                <mx:Object Product="Flash" Code="1000"/>
                <mx:Object Product="Flex" Code="2000"/>
                <mx:Object Product="ColdFusion" Code="3000"/>
                <mx:Object Product="JRun" Code="4000"/>
            </mx:dataProvider>
        </mx:DataGrid>
        <mx:Button id="myButton"
            label="Print"
            click="doPrint();"/>
    </mx:Form>
</mx:Application>
```

The following MyPrintView.mxml file defines the component used by the application's `doPrint()` method. The component is a VBox container; it contains a Label control, into which the application writes the contact information from the first three fields of the form, and a PrintDataGrid control, which displays the data from the data source of the screen view's DataGrid control. For more information on the PrintDataGrid control and its advantages for printing, see "Using the PrintDataGrid control for multipage grids" on page 1172.

```
<?xml version="1.0"?>
<!-- printing\myComponents\MyPrintView.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="#FFFFFF"
    height="250" width="450"
    paddingTop="50" paddingLeft="50" paddingRight="50">

    <!-- The controls to print, a label and a PrintDataGrid control. -->
    <mx:Label id="contact"/>
    <mx:PrintDataGrid id="myDataGrid" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="Product"/>
            <mx:DataGridColumn dataField="Code"/>
        </mx:columns>
    </mx:PrintDataGrid>
</mx:VBox>
```

Printing multipage output

The topics in this section describe how you can print well-formatted multipage output under the following condition:

- When each control fits on a print page or less. You often encounter such jobs when printing a form with fixed-length fields.
- When the printed output includes one or more PrintDataGrid controls that are too long to print on a single page, particularly if the control height might vary, depending on the data. A good example of this type of output is a customer order receipt, which starts with the customer information, has an indeterminate number of order line items, and ends with total information.

Printing known-length multipage output

If you know the length of each component in a multipage document, you can create a separate print layout component for each page you print, and specify each layout page in a separate `addObject()` method, as follows:

```
printJob.addObject(introPrintView, "ShowAll");
printJob.addObject(finDetailPrintView, "ShowAll");
```

```
printJob.addObject(hrDetailPrintView, "ShowAll");
printJob.addObject(summaryPrintView, "ShowAll");
```

Using the PrintDataGrid control for multipage grids

When a DataGrid control with many rows does not fit on a single screen in your application, you typically have scroll bars that let users view all the data. When you print the DataGrid control, the output is the same as the screen display. Therefore, if your DataGrid control has rows or columns that are not immediately visible, they do not print. If you replace the DataGrid control with a PrintDataGrid control that does not have a height specified (or has a large height), you print all the rows, but some rows could be partially printed on the bottom of one page and partially printed at the top of another, as you often see with HTML printout.

You can solve these problems by using the following features of the PrintDataGrid control. These features let you correctly print grids that contain multiple pages of data without splitting rows across pages.

sizeToPage property Makes the printed data grid contain only full rows.

nextPage() method Gets the next printable page of data.

validNextPage property Is `true` if printing the data requires an additional page.

Using the sizeToPage attribute to format pages

A PrintDataGrid page consists of the rows that are visible in the control's current view. Suppose, for example, that a PrintDataGrid control has a height of 130 pixels. The total height of each row and header is 30 pixels, and the control's data provider has 10 rows. In this situation, the printed PrintDataGrid page contains only three complete data rows, plus the header. The `sizeToPage` property specifies whether to include a fourth and partial data row.

The `sizeToPage` property, which is `true` by default, causes the PrintDataGrid control to remove any partially visible or empty rows and to resize itself to include only complete rows in the current view. For the data grid described in the preceding paragraph, when this property is `true`, the DataGrid shrinks to show three complete data rows, and no incomplete rows; if the attribute is `false`, the grid includes a partial row at the bottom.

The following properties provide information on page sizing that are affected by the `sizeToPage` property:

Property	Description
`currentPageHeight`	Contains the height of the grid, in pixels, that results if the `sizeToPage` property is `true`. If the `sizeToPage` property is `true`, the `currentPageHeight` property equals the `height` property.
`originalHeight`	Contains the grid height that results if the `sizeToPage` property is `false`. If the `sizeToPage` property is `false`, the `originalHeight` property equals the `height` property.

In most applications, you leave the `sizeToPage` attribute at its default value (`true`), and use the `height` property to determine the grid height.

The `sizeToPage` property does *not* affect the way the page breaks when a single PrintDataGrid control page is longer than a print page. To print multipage data grids without splitting rows, you must divide the grid items into multiple views by using the `nextPage()` method, as described in "Using the nextPage() method and validNextPage property to print multiple pages" on page 1173.

Using the nextPage() method and validNextPage property to print multiple pages

The `validNextPage` property is `true` if the PrintDataGrid control has data beyond the rows that fit on the current print page. You use it to determine whether you need to format and print an additional page.

The `nextPage()` method lets you page through the data provider contents by setting the first row of the PrintDataGrid control to be the data provider row that follows the last row of the previous PrintDataGrid page. In other words, the `nextPage()` method increases the grid's `verticalScrollPosition` property by the value of the grid's `rowCount` property.

The following code shows a loop that prints a grid using multiple pages, without having rows that span pages:

```
// Queue the first page.
printJob.addObject(thePrintView);
// While there are more pages, print them.
while (thePrintView.myDataGrid.validNextPage) {
  //Put the next page of data in the view.
  thePrintView.myDataGrid.nextPage();
  //Queue the additional page.
  printJob.addObject(thePrintView);
}
```

The section "Example: Printing with multipage PrintDataGrid controls" on page 1174 shows how to use the `nextPage()` method to print a report with a multipage data grid.

Updating the PrintDataGrid layout

When you use a PrintDataGrid control to print a single data grid across multiple pages, you queue each page of the grid individually. If your application customizes each page beyond simply using the `nextPage()` method to page through the PrintDataGrid, you must call the `validateNow()` method to update the page layout before you print each page, as shown in "Print output component" on page 1178.

Example: Printing with multipage PrintDataGrid controls

The following example prints a data grid in which you can specify the number of items in the data provider. You can, therefore, set the DataGrid control contents to print on one, two, or more pages, so that you can see the effects of different-sized data sets on the printed result.

The example also shows how you can put header information before the grid and footer information after the grid, as in a shipping list or receipt. It uses the technique of selectively showing and hiding the header and footer, depending on the page being printed. To keep the code as short as possible, the example uses simple placeholder information only.

The application consists of the following files:

- The application file displays the form to the user, including TextArea and Button controls to set the number of lines and a Print button. The file includes the code to initialize the view, get the data, and handle the user's print request. It uses the FormPrintView MXML component as a template for the printed output.
- The FormPrintView.mxml file formats the printed output. It has two major elements:

  Print output template Includes the PrintDataGrid control and uses two MXML components to format the header and footer contents.

  showPage() function Determines which sections of the template to include in a particular page of the output, based on the page type: first, middle, last, or single. On the first page of multipage output, the `showPage()` function hides the footer; on the middle and last pages, it hides the header. On a single page, it shows both header and footer.
- The FormPrintHeader.mxml and formPrintFooter.mxml files specify the contents of the start and the end of the output. To keep the application simple, the header has a single, static Label control. The footer displays a total of the numbers in the Quantity column. In a more complete application, the header page could have, for example, a shipping address, and the footer page could show more detail about the shipment totals.

The files include detailed comments explaining the purpose of the code.

Multipage print application file

The following code shows the multipage print application file:

```
<?xml version="1.0"?>
<!-- printing\MultiPagePrint.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initData();">

    <mx:Script>
    <![CDATA[
        import mx.printing.*;
        import mx.collections.ArrayCollection;
        // Import the MXML custom print view control.
        import myComponents.FormPrintView;

        // Declare variables and initialize simple variables.
        // The dgProvider ArrayCollection is the DataGrid data provider.
        // It must be bindable because you change its contents dynamically.
        [Bindable]
        public var dgProvider:ArrayCollection;
        public var footerHeight:Number = 20;
        public var prodIndex:Number;
        public var prodTotal:Number = 0;

        // Data initialization, called when the application initializes.
        public function initData():void {
            // Create the data provider for the DataGrid control.
            dgProvider = new ArrayCollection;
        }

        // Fill the dgProvider ArrayCollection with the specified items.
        public function setdgProvider(items:int):void {
            // First initialize the index and clear any existing data.
            prodIndex=1;
            dgProvider.removeAll();

            // Fill the ArrayCollection, and calculate a product total.
            // For simplicity, it increases the Index field value by
            // 1, and the Qty field by 7 for each item.
            for (var z:int=0; z<items; z++)
            {
                var prod1:Object = {};
                prod1.Qty = prodIndex * 7;
                prod1.Index = prodIndex++;
                prodTotal += prod1.Qty;
                dgProvider.addItem(prod1);
            }
```

```
}

// The function to print the output.
public function doPrint():void {
    // Create a FlexPrintJob instance.
    var printJob:FlexPrintJob = new FlexPrintJob();

    // Start the print job.
    if (printJob.start()) {
        // Create a FormPrintView control
        // as a child of the application.
        var thePrintView:FormPrintView = new FormPrintView();
        addChild(thePrintView);

        // Set the print view properties.
        thePrintView.width=printJob.pageWidth;
        thePrintView.height=printJob.pageHeight;
        thePrintView.prodTotal = prodTotal;

        // Set the data provider of the FormPrintView
        // component's DataGrid to be the data provider of
        // the displayed DataGrid.
        thePrintView.myDataGrid.dataProvider =
            myDataGrid.dataProvider;

        // Create a single-page image.
        thePrintView.showPage("single");

        // If the print image's DataGrid can hold all the
        // data provider's rows, add the page to the print job.
        if(!thePrintView.myDataGrid.validNextPage)
        {
            printJob.addObject(thePrintView);
        }
        // Otherwise, the job requires multiple pages.
        else
        {
            // Create the first page and add it to the print job.
            thePrintView.showPage("first");
            printJob.addObject(thePrintView);
            thePrintView.pageNumber++;

            // Loop through the following code
            // until all pages are queued.
            while(true)
            {
                // Move the next page of data to the top of
                // the PrintDataGrid.
                thePrintView.myDataGrid.nextPage();
```

```
                    // Try creating a last page.
                    thePrintView.showPage("last");

                    // If the page holds the remaining data, or if
                    // the last page was completely filled by the last
                    // grid data, queue it for printing.
                    // Test if there is data for another
                    // PrintDataGrid page.
                    if(!thePrintView.myDataGrid.validNextPage)
                    {
                        // This is the last page;
                        // queue it and exit the print loop.
                        printJob.addObject(thePrintView);
                        break;
                    }
                    else
                    // This is not the last page. Queue a middle page.
                    {
                        thePrintView.showPage("middle");
                        printJob.addObject(thePrintView);
                        thePrintView.pageNumber++;
                    }
                }
            }
            // All pages are queued; remove the FormPrintView
            // control to free memory.
            removeChild(thePrintView);
        }
        // Send the job to the printer.
        printJob.send();
    }
]]>
</mx:Script>

<!-- The form that appears on the user's system.-->
<mx:Form id="myForm" width="80%">
    <mx:FormHeading label="Product Information"/>
            <mx:DataGrid id="myDataGrid" dataProvider="{dgProvider}">
            <mx:columns>
                <mx:DataGridColumn dataField="Index"/>
                <mx:DataGridColumn dataField="Qty"/>
            </mx:columns>
        </mx:DataGrid>
    <mx:Text width="100%"
        text="Specify the number of lines and click Fill Grid first.
        Then you can click Print."/>
    <mx:TextInput id="dataItems" text="35"/>
    <mx:HBox>
        <mx:Button id="setDP"
            label="Fill Grid"
```

```
                click="setdgProvider(int(dataItems.text));"/>
            <mx:Button id="printDG"
                label="Print"
                click="doPrint();"/>
        </mx:HBox>
    </mx:Form>
</mx:Application>
```

Print output component

The following lines show the FormPrintView.mxml custom component file:

```
<?xml version="1.0"?>
<!-- printing\myComponents\FormPrintView.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*"
    backgroundColor="#FFFFFF"
    paddingTop="50" paddingBottom="50" paddingLeft="50">

    <mx:Script>
        <![CDATA[
            import mx.core.*

            // Declare and initialize the variables used in the component.
            // The application sets the actual prodTotal value.
            [Bindable]
            public var pageNumber:Number = 1;
            [Bindable]
            public var prodTotal:Number = 0;

           // Control the page contents by selectively hiding the header and
            // footer based on the page type.
            public function showPage(pageType:String):void {
                if(pageType == "first" || pageType == "middle") {
                    // Hide the footer.
                    footer.includeInLayout=false;
                    footer.visible = false;
                }
                if(pageType == "middle" || pageType == "last") {
                    // The header won't be used again; hide it.
                    header.includeInLayout=false;
                    header.visible = false;
                }
                if(pageType == "last") {
                    // Show the footer.
                    footer.includeInLayout=true;
                    footer.visible = true;
                }
                //Update the DataGrid layout to reflect the results.
                validateNow();
```

```
            }
        ]]>
    </mx:Script>

    <!-- The template for the printed page,
        with the contents for all pages. -->
    <mx:VBox width="80%" horizontalAlign="left">
        <mx:Label text="Page {pageNumber}"/>
    </mx:VBox>
    <MyComp:FormPrintHeader id="header"/>

    <!-- The sizeToPage property is true by default, so the last
        page has only as many grid rows as are needed for the data. -->
    <mx:PrintDataGrid id="myDataGrid" width="60%" height="100%">
    <!-- Specify the columns to ensure that their order is correct. -->
        <mx:columns>
            <mx:DataGridColumn dataField="Index" />
            <mx:DataGridColumn dataField="Qty" />
        </mx:columns>
    </mx:PrintDataGrid>

    <!-- Create a FormPrintFooter control
        and set its prodTotal variable. -->
    <MyComp:FormPrintFooter id="footer" pTotal="{prodTotal}"/>
</mx:VBox>
```

Header and footer files

The following lines show the FormPrintHeader.mxml file:

```
<?xml version="1.0"?>
<!-- printing\myComponents\FormPrintHeader.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    width="60%"
    horizontalAlign="right" >

    <mx:Label text="This is a placeholder for first page contents"/>
</mx:VBox>
```

The following lines show the FormPrintFooter.mxml file:

```
<?xml version="1.0"?>
<!-- printing\myComponents\FormPrintFooter.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    width="60%"
    horizontalAlign="right">

    <!-- Declare and initialize the product total variable. -->
    <mx:Script>
        <![CDATA[
            [Bindable]
            public var pTotal:Number = 0;
        ]]>
        </mx:Script>

    <mx:Label text="Product Total: {pTotal}"/>
</mx:VBox>
```

CHAPTER 34

Communicating with the Wrapper

34

This topic describes how to exchange data between an Adobe Flex application and the HTML page that embeds that application.

Contents

About exchanging data with Flex applications

Flex applications generally exist inside larger web applications that control everything from security to state management to the overall look and feel of the website. In this scenario it is important that the Flex application is able to communicate with the surrounding environment, providing a deeper integration with the larger web application. Enabling the Flex application to communicate with the environment provides a method of integration with other technologies such as AJAX.

Often, a Flex application is loaded in a browser within a wrapper. This wrapper is often an HTML page that can include JavaScript or other client-side logic that the Flex application can interact with. For more information about the wrapper, see Chapter 16, "Creating a Wrapper," in *Building and Deploying Flex 2 Applications*.

There are several ways to communicate between the surrounding environment and the Flex application; depending on the type of integration required, any combination of `flashVars` properties, query string parameters, the `navigateToURL()` method, and with the ExternalInterface class can be employed.

To pass request data into your Flex applications, use query string parameters and access the value using either the Application.application.parameters or LoaderConfig.parameters objects. You can also pass data as `flashVars` properties in the wrapper and access the values using the `parameters` property of the Application and LoaderConfig objects. For more information, see "Using flashVars" on page 1188. Using these techniques, you can personalize a Flex application without triggering a recompilation.

Use the methods of the ExternalInterface API to call the methods of your Flex applications and vice versa. The `addCallback()` method exposes methods of your Flex application to the wrapper. The `call()` method invokes a method within the wrapper and returns any results. If the wrapper is HTML, the `addCallback()` and `call()` methods enable method invocation between your Flex application and the hosted JavaScript running within the browser. For more information, see "About the ExternalInterface API" on page 1184.

In some situations, you want to open a new browser window or navigate to a new location. You can do this with the `navigateToURL()` global function. Although it is not part of the ExternalInterface API, this method is flexible enough to let you write JavaScript inside it, and invoke JavaScript functions on the resulting HTML page. The `navigateToURL()` method is a global function in the flash.net package.

Accessing environment information

Flex provides some simple mechanisms for getting information about the browser and the environment in which the application runs. From within the Flex application, you can get the URL to the SWF file; you can also get the context root. The following example gets the SWF file's URL, constructs the host name from the URL, and displays the context root, which is accessible with the `@ContextRoot()` token:

```
<?xml version="1.0"?>
<!-- wrapper/GetURLInfo.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="getHostName()">
  <mx:Script><![CDATA[

  [Bindable]
  public var g_HostString:String;
  [Bindable]
  public var g_ContextRoot:String;
  [Bindable]
  public var g_BaseURL:String;

  private function getHostName():void {
     g_BaseURL = Application.application.url;
     var pattern1:RegExp = new RegExp("http://[^/]*/");
     if (pattern1.test(g_BaseURL) == true) {
        g_HostString = pattern1.exec(g_BaseURL).toString();
     } else{
        g_HostString = "http://localhost/"
     }
  }
  ]]></mx:Script>

  <mx:Form>
     <mx:FormItem label="Base URL:">
        <mx:Label text="{g_BaseURL}"/>
     </mx:FormItem>
     <mx:FormItem label="Host Name:">
        <mx:Label text="{g_HostString}"/>
     </mx:FormItem>
     <mx:FormItem label="Context Root:">
        <mx:Label text="@ContextRoot()"/>
     </mx:FormItem>
  </mx:Form>
</mx:Application>
```

You can also use the flash.system.Capabilities class to access information about the client, such as Operating System, Player version, and language. For more information, see *Using Flash ActionScript*.

You can access more information about the browser and the application's environment using the ExternalInterface. For more information, see "About the ExternalInterface API" on page 1184.

Enabling Netscape connections

In Netscape browsers, the `<embed>` tag in your wrapper must include `swliveconnect=true` to allow communication between the browser and your Flex application. This lets the Flex application connect with the page's scripting language (usually JavaScript). You add this parameter to the `<embed>` tags in the Flex application's wrapper, as the following example shows:

```
<embed pluginspage='http://www.macromedia.com/go/getflashplayer'
  width='300'
  height='100'
  flashvars=''
  src='TitleTest.mxml.swf'
  name='MyApp'
  SWLIVECONNECT='true'
/>
```

You are not required to set the value of `swliveconnect` to `true` in the `<object>` tag because the `<object>` tag is used by Microsoft Internet Explorer, but not Netscape browsers.

About the ExternalInterface API

You use the ExternalInterface API to let your Flex application call methods in the wrapper and to allow the wrapper to call functions in your Flex application. The ExternalInterface API consists primarily of the `call()` and `addCallback()` methods in the flash.external package.

The following browsers support the ExternalInterface API:

- All versions of Internet Explorer for Windows (5.0 and later)
- Embedded custom ActiveX containers, such as a desktop application embedding Adobe Flash Player ActiveX control
- Any browser that supports the NPRuntime interface (which currently includes the following browsers):
  - Firefox 1.0 and later
  - Mozilla 1.7.5 and later
  - Netscape 8.0 and later
  - Safari 1.3 and later

Before you execute code that uses the ExternalInterface API, you should check whether the browser supports it. You do this by using the `available` property of the ExternalInterface object in your Flex application. The `available` property is a Boolean value that is `true` if the browser supports the ExternalInterface API and `false` if the browser does not. It is a read-only property.

The following example uses the `available` property to detect support for the ExternalInterface API before executing methods that use the class:

```
<?xml version="1.0"?>
<!-- wrapper/CheckExternalInterface.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="checkEI()">
  <mx:Script><![CDATA[
     [Bindable]
     public var eiStatus:String;

     private function checkEI():void {
           eiStatus = ExternalInterface.available.toString();
     }

  ]]></mx:Script>

  <mx:Label id="l1" text="{eiStatus}"/>

</mx:Application>
```

For examples of using the ExternalInterface API with Flex applications, see "Using the ExternalInterface API to access JavaScript from Flex" on page 1192 and "Accessing Flex from JavaScript" on page 1205.

In addition to requiring that browsers meet certain version requirements, the ExternalInterface API requires that JavaScript is enabled in the browser. You can use the `<noscript>` tag in the HTML page to handle a browser with disabled JavaScript. For more information, see "Handling browsers that disable JavaScript" on page 1211.

The `available` property determines only if the browser can support the ExternalInterface API, based on its version and manufacturer. If JavaScript is disabled in the browser, the `available` property still returns `true`.

The ExternalInterface API is restricted by the security sandbox in which the SWF file is running. Its use relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` parameters define. You set the values of the `allowScriptAccess` and `allowNetworking` parameters in the SWF file's wrapper.

For more information on these parameters, see Chapter 16, "Creating a Wrapper," in *Building and Deploying Flex 2 Applications*.

For more information on security restrictions, see "About ExternalInterface API security in Flex" on page 1211; also see Chapter 4, "Applying Flex Security," in *Building and Deploying Flex 2 Applications*.

Passing request data to Flex applications

You can pass request data to Flex applications by defining `flashVars` properties in the wrapper. This is described in "Using flashVars" on page 1188. If your users request MXML files from the Adobe Flex Data Services server directly, Flex converts query string parameters to `flashVars` variables for you. This is described in "Using query string parameters" on page 1191.

Changing the query string parameter or the `flashVars` property does not trigger a recompilation of the Flex application if you are using the Flex Data Services server.

To use `flashVars` or query string parameters inside your MXML file, you access the Application.application.parameters object, as described in "Using the Application.application.parameters object" on page 1186.

Using the Application.application.parameters object

The `parameters` property of the Application object points to a dynamic object that stores parameters as name-value pairs. You can access variables on the parameters object by specifying parameters.*variable_name*.

The following example defines the `myName` and `myHometown` parameters and binds them to the text of Label controls:

```
<?xml version="1.0"?>
<!-- wrapper/ApplicationParameters.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initVars()">
  <mx:Script><![CDATA[
     // Declare bindable properties in Application scope.
     [Bindable]
     public var myName:String;
     [Bindable]
     public var myHometown:String;

     // Assign values to new properties.
     private function initVars():void {
        myName = Application.application.parameters.myName;
        myHometown = Application.application.parameters.myHometown;
     }
  ]]></mx:Script>

  <mx:VBox>
  <mx:HBox>
     <mx:Label text="Name: "/>
     <mx:Label text="{myName}" fontWeight="bold"/>
  </mx:HBox>
  <mx:HBox>
     <mx:Label text="Hometown: "/>
     <mx:Label text="{myHometown}" fontWeight="bold"/>
  </mx:HBox>
  </mx:VBox>
</mx:Application>
```

When a user requests this application with the `myName` and `myHometown` parameters defined as `flashVars` variables, Flex displays their values in the Label controls.

To view all the `flashVars` properties, you can iterate over the parameters object, as the following example shows:

```
<?xml version="1.0"?>
<!-- wrapper/IterateOverApplicationParameters.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     private function init():void {
        for (var i:String in Application.application.parameters) {
           ta1.text += i + ":" + Application.application.parameters[i] +
"\n";
        }
     }
  ]]></mx:Script>

  <mx:TextArea id="ta1" width="300" height="200"/>

</mx:Application>
```

Flex does not recompile the application if the request variables have changed. As a result, if you dynamically set the values of the `flashVars` properties or query string parameters, you do not force a recompilation.

Query string parameters must be URL encoded. The format of the string is a set of name-value pairs separated by an ampersand (&). You can escape special and nonprintable characters with a percent symbol (%) followed by a two-digit hexadecimal value. You can represent a single blank space using the plus sign (+).

The encoding for `flashVars` properties and query string parameters is the same as the page. Internet Explorer provides UTF-16-compliant strings on the Windows platform. Netscape sends a UTF-8-encoded string to Adobe Flash Player.

Most browsers support a `flashVars` String or query string up to 64 KB (65535 bytes) in length. They can include any number of name-value pairs.

Using flashVars

You can pass variables to your Flex applications using the `flashVars` properties in the `<object>` and `<embed>` tags in your wrapper.

The following example sets the values of the `firstname`, `middlename`, and `lastname` `flashVars` properties inside the `<object>` tag in a simple wrapper:

```
<html>
<head>
<title>/flex2/code/wrapper/SimplestFlashVarTestWrapper.html</title>
<style>
body { margin: 0px;
 overflow:hidden }
</style>
</head>
<body scroll='no'>
<table width='100%' height='100%' cellspacing='0' cellpadding='0'><tr><td
valign='top'>

<h1>Simplest FlashVarTest Wrapper</h1>

<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='100%' width='100%'>
<param name='src' value='FlashVarTest.swf'/>
<param name='flashVars' value='firstName=Nick&lastName=Danger'/>
<embed name='mySwf' src='FlashVarTest.swf' pluginspage='http://
www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars='firstName=Nick&lastName=Danger'/>
</object>

</td></tr></table>
</body>
</html>
```

If you are using the wrapper that is generated by Flex Data Services or wrappers that are included in the resources/html-templates directory, your wrapper might not look the same, but the basic approach to passing the `flashVars` variable is. For example, you might insert `flashVars` variables by appending them to a function parameter, as the following example shows:

```
"flashvars","historyUrl=%2Fflex%2Fflex%2Dinternal%3Faction%3Dhistory%5Fhtml
  &lconid=" + lc_id + "&firstName=Nick&lastName=Danger",
```

The value of the `flashVars` variable does not have to be static. If you use JSP to return the wrapper, for example, you can use any JSP expression for the value of the `flashVars` variable that can be evaluated to a String.

The following example uses the values stored in the HttpServletRequest object (in this case, you can use form or query string parameters):

```
<html>
<head>
<title>/flex2/code/wrapper/DynamicFlashVarTestWrapper.jsp</title>
<style>
body { margin: 0px;
 overflow:hidden }
</style>
</head>

<%
String fName = (String) request.getParameter("firstname");
String mName = (String) request.getParameter("middlename");
String lName = (String) request.getParameter("lastname");
%>

<body scroll='no'>
<table width='100%' height='100%' cellspacing='0' cellpadding='0'><tr><td
valign='top'>

<script>
<h1>Dynamic FlashVarTest Wrapper</h1>
</script>


<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='100%' width='100%'>
<param name='src' value='../assets/FlashVarTest.swf'/>
<param name='flashVars' value='firstname=<%= fName %>&lastname=<%= lName
%>'/>
<embed name='mySwf' src='../assets/FlashVarTest.swf' pluginspage='http://
www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars='firstname=<%= fName %>&lastname=<%= lName %>'/>
</object>

</td></tr></table>
</body>
</html>
```

Using query string parameters

When using Flex Data Services, you can add query string parameters to the client's request string, and use the values of those parameters in any Flex application. In effect, Flex supports GET request variables passed in as name-value pairs in the URL.

When creating the wrapper, Flex Data Services converts query string parameters to a `flashVars` variable. As a result, you can access query string parameters in your Flex application if you are using Flex Data Services or if you implement server-side logic that handles generating a wrapper. For example, suppose you are using the web-tier compiler, and you request the following query string in the browser's address bar:

```
http://localhost:8101/flex/charts/PieChart1.mxml?fname=Nick&lname=Danger
```

Flex adds those values as `flashVars` variables to the wrapper.

NOTE	You cannot pass query string parameters to a Flex application that runs inside the stand-alone Flash Player.

You can also append query string parameters to the `<object>` and `<embed>` tags' `src` properties in the wrapper. The `src` property identifies the location of the Flex application's SWF file. You access these the same way that you access query string parameters in the browser's address bar.

The following example appends query string parameters to the `src` properties in the custom wrapper:

```
<object ... >
  <param name='src' value='TitleTest.mxml.swf?myName=Danger'>
  ...
  <embed src='TitleTest.mxml.swf?myName=Danger' ... />
</object>
```

As with `flashVars` properties, you access the value of the query string parameters in your Flex application on the parameters object:

```
var myName:String = Application.application.parameters.myName;
```

If your user requests the SWF file directly, without a wrapper, you can access variables on the query string without providing additional code. The following URL passes the name Nick and the hometown San Francisco to the Flex application:

```
http://localhost:8100/flex/myApp.swf?myName=Nick&myHometown=San%20Francisco
```

Accessing JavaScript functions from Flex

You can call JavaScript functions in the enclosing HTML page from your Flex application. By allowing Flex to communicate with the browser, you can change style settings, invoke remote processes, or perform any other action that you can normally do from within the page's scripts.

You can even pass data from the Flex application to the enclosing HTML page, process it, and then pass it back to the Flex application. You do this by using the ExternalInterface API or the `navigateToURL()` method. For more information, see "Using the ExternalInterface API to access JavaScript from Flex" on page 1192 and "Using the navigateToURL() method in Flex" on page 1199.

Whenever you communicate with the enclosing page, you must determine if the browser can handle the kinds of actions that you want to perform. Therefore, you should first determine if the browser supports the objects that you want to use. For some general guidelines for determining browser support, see "Using the ExternalInterface API to access JavaScript from Flex" on page 1192.

Using the ExternalInterface API to access JavaScript from Flex

The easiest way to call JavaScript functions from your Flex application is to use the ExternalInterface API. You can use this API to call any JavaScript method on the wrapper, pass parameters, and get a return value. If the method call fails, Flex returns an exception.

The ExternalInterface API encapsulates checks for browser support, so you are not required to do that when using its methods. However, you can check whether the browser supports the interface by using its `available` property. For more information, see "About the ExternalInterface API" on page 1184.

The ExternalInterface API consists of a single class, flash.external.ExternalInterface. This class has the `call()` static method that you use to facilitate the JavaScript to Flash communication. You can also call methods in your Flex application from the wrapper by using the ExternalInterface API's `addCallback()` method. For more information, see "About the addCallback() method" on page 1212.

To use the ExternalInterface, you must make the `id` and `name` properties in the HTML page callable. For more information, see "Editing the Flex application's id and name properties" on page 1210.

Calling JavaScript methods from Flex applications

The ExternalInterface API makes it very simple to call methods in the enclosing wrapper. You use the static `call()` method, which has the following signature:

```
flash.external.ExternalInterface.call(function_name:
  String[, arg1, ...]):Object;
```

The *`function_name`* is the name of the function in the HTML page's JavaScript. The arguments are the arguments that you pass to the JavaScript function. You can pass one or more arguments in the traditional way of separating them with commas, or you can pass an object that is deserialized by the browser. The arguments are optional.

The following example script block calls the JavaScript `changeDocumentTitle()` function in the enclosing wrapper by using the `call()` method:

```
<?xml version="1.0"?>
<!-- wrapper/WrapperCaller.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
     import flash.external.*;

     public function callWrapper():void {
        var s:String;
        if (ExternalInterface.available) {
           var wrapperFunction:String = "changeDocumentTitle";
           s = ExternalInterface.call(wrapperFunction,ti1.text);
        } else {
           s = "Wrapper not available";
        }
        trace(s);
     }
  </mx:Script>

  <mx:Form>
     <mx:FormItem label="New Title:">
        <mx:TextInput id="ti1"/>
     </mx:FormItem>
  </mx:Form>

  <mx:Button label="Change Document Title" click="callWrapper()"/>
</mx:Application>
```

On your HTML page, you define a function as you would any other JavaScript function. You can return a value, as the following example shows:

```
<html><head>
<title>wrapper/WrapperBeingCalled.html</title>
</head>
<body scroll='no'>

<SCRIPT LANGUAGE="JavaScript">
function changeDocumentTitle(a) {
window.document.title=a;
alert(a);
return "successful";
}
</SCRIPT>

<h1>Wrapper Being Called</h1>
<table width='100%' height='100%' cellspacing='0' cellpadding='0'>
<tr>
<td valign='top'>
<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='200' width='400'>
<param name='src' value='WrapperCaller.swf'/>
<param name='flashVars' value=''/>
<embed name='mySwf' src='WrapperCaller.swf' pluginspage='http://
www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars=''/>
</object>
</td>
</tr>
</table>
</body></html>
```

This feature requires that the embedded movie file have an `id` attribute. Without it, no call from your Flex application will succeed.

The `call()` method accepts zero or more arguments, which can be ActionScript types. Flex serializes the ActionScript types as JavaScript numbers and strings. If you pass an object, you can access the properties of that deserialized object in the JavaScript, as the following example shows:

```
<?xml version="1.0"?>
<!-- wrapper/DataTypeSender.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import flash.external.*;

     public function callWrapper():void {
        var s:String;
        if (ExternalInterface.available) {
           var o:Object = new Object();
           o.fname = "Nick";
           o.lname = "Danger";
           var wrapperFunction:String = "receiveComplexDataTypes";
           s = ExternalInterface.call(wrapperFunction, o);
        } else {
           s = "Wrapper not available";
        }
        trace(s);
     }
  ]]></mx:Script>

  <mx:Button label="Send" click="callWrapper()"/>

</mx:Application>
```

Flex only serializes public, nonstatic variables and read-write properties of ActionScript objects. You can pass numbers and strings as properties on objects, simple objects such as primitive types and arrays, or arrays of simple objects.

The JavaScript code can then access properties of the object, as the following example shows:

```
<html><head>
<title>wrapper/DataTypeWrapper.html</title>
</head>
<body scroll='no'>

<SCRIPT LANGUAGE="JavaScript">
function receiveComplexDataTypes(o) {
alert("Welcome " + o.fname + " " + o.lname + "!");
return "successful";
}
</SCRIPT>

<h1>Data Type Wrapper</h1>

<table width='100%' height='100%' cellspacing='0' cellpadding='0'>
<tr>
<td valign='top'>
<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='200' width='400'>
<param name='src' value='DataTypeSender.swf'/>
<param name='flashVars' value=''/>
<embed name='mySwf' src='DataTypeSender.swf' pluginspage='http://
www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars=''/>
</object>
</td>
</tr>
</table>

</body></html>
```

You can also embed objects within objects, such as an Array within an Object, as the following example shows. Add the following code in your Flex application's `<mx:Script>` block:

```
<?xml version="1.0"?>
<!-- wrapper/ComplexDataTypeSender.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import flash.external.*;

     public function callWrapper():void {
        var s:String;
        if (ExternalInterface.available) {
           var o:Object = new Object();
           o.fname = "Nick";
           o.lname = "Danger";
           o.b = new Array("DdW","E&T","LotR:TS");
           var wrapperFunction:String = "receiveComplexDataTypes";
           s = ExternalInterface.call(wrapperFunction, o);
        } else {
           s = "Wrapper not available";
        }
        trace(s);
     }
  ]]></mx:Script>

  <mx:Button label="Send" click="callWrapper()"/>

</mx:Application>
```

The code triggers the JavaScript function in the following wrapper:

```
<html><head>
<title>wrapper/ComplexDataTypeWrapper.html</title>
</head>
<body scroll='no'>

<SCRIPT LANGUAGE="JavaScript">
function receiveComplexDataTypes(o) {
// Get value of fname and lname properties.
var s = ("Welcome " + o.fname + " " + o.lname + "!\n");
// Iterate over embedded object's properties.
for (i=0; i<o.b.length; i++) {
s +=  o.b[i] + "\n";
}
alert(s);
}
</SCRIPT>

<h1>Complex Data Type Wrapper</h1>

<table width='100%' height='100%' cellspacing='0' cellpadding='0'>
<tr>
<td valign='top'>
<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='200' width='400'>
<param name='src' value='ComplexDataTypeSender.swf'/>
<param name='flashVars' value=''/>
<embed name='mySwf' src='ComplexDataTypeSender.swf' pluginspage='http://
www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars=''/>
</object>
</td>
</tr>
</table>

</body></html>
```

Flex and Flash Player have strict security in place to prevent cross-site scripting. By default, you cannot call script on an HTML page if the HTML page is not in the same domain as the Flex application. However, you can expand the sources from which scripts can be called. For more information, see "About ExternalInterface API security in Flex" on page 1211.

You cannot pass objects or arrays that contain circular references. For example, you cannot pass the following object:

```
var obj = new Object();
obj.prop = obj; // Circular reference.
```

Circular references cause infinite loops in both ActionScript and JavaScript.

Using the navigateToURL() method in Flex

The `navigateToURL()` method loads a document from a specific URL into a window or passes variables to another application at a defined URL. You can use this method to call JavaScript functions in the HTML page that encloses a Flex application.

You should not confuse the functionality of the `navigateToURL()` method with the `load()` method of the URLLoader class. The URLLoader class loads a specified URL into an object for manipulation with ActionScript. The `navigateToURL()` method navigates to the specified URL with a browser.

In most cases, you should use the ExternalInterface API to perform Flex-to-wrapper communication. However, the `navigateToURL()` method is not part of the ExternalInterface API and, therefore, does not have as stringent a set of requirements for which browsers support it. These requirements are described in "About the ExternalInterface API" on page 1184.

The `navigateToURL()` method is restricted by the security sandbox in which the SWF file is running. Its use relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` parameters define. You set the values of the `allowScriptAccess` and `allowNetworking` parameters in the SWF file's wrapper.

For more information on these parameters, see Chapter 16, "Creating a Wrapper," in *Building and Deploying Flex 2 Applications*. For more information on security restrictions, see Chapter 4, "Applying Flex Security," in *Building and Deploying Flex 2 Applications*.

The navigateToURL() method syntax

The `navigateToURL()` method is in the flash.net package. It has the following signature:

```
navigateToURL(request:URLRequest, window:String):void
```

The *`request`* argument is a URLRequest object that specifies the destination. The *`window`* argument specifies whether to launch a new browser window or load the new URL into the current window. The following table describes the valid values for the *`window`* argument:

Value	Description
`_self`	Specifies the current frame in the current window.
`_blank`	Specifies a new window. This new window acts as a pop-up window in the client's browser, so you must be aware that a pop-up blocker could prevent it from loading.

Value	Description
_parent	Specifies the parent of the current frame.
_top	Specifies the top-level frame in the current window.

You pass a URLRequest object to the navigateToURL() method. This object defines the URL target, variables, method (POST or GET), window, and headers for the request. The following example defines a simple URL to navigate to:

```
<?xml version="1.0"?>
<!-- wrapper/SimplestNavigateToURL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import flash.net.*;
     public function openNewWindow(event:MouseEvent):void {
        var u:URLRequest = new URLRequest("http://www.adobe.com/flex");
        navigateToURL(u,"_blank");
     }
  ]]></mx:Script>
  <mx:Button label="Open New Window" click="openNewWindow(event)"/>
</mx:Application>
```

The navigateToURL() method URL encodes the value of the url argument.

To send data with a URLRequest object, you can append variables to the request string. The following example launches a new window and passes a search term to the URL:

```
<?xml version="1.0"?>
<!-- wrapper/SimpleNavigateToURL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import flash.net.*;
     public function executeSearch(event:MouseEvent):void {
        var u:URLRequest = new URLRequest("http://www.google.com/
search?hl=en&q=" + ta1.text);
        navigateToURL(u,"_blank");
     }
  ]]></mx:Script>
  <mx:TextArea id="ta1"/>
  <mx:Button label="Search" click="executeSearch(event)"/>
</mx:Application>
```

In addition to appending strings, you can also use the `data` property of the URLRequest to add URL variables to a GET or POST request. The query string parameters are of type URLVariables. Flex adds ampersand delimiters for you in the URL request string. The following example adds `name=fred` to the URLRequest:

```
<?xml version="1.0"?>
<!-- wrapper/NavigateWithGetMethod.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import flash.net.*;
     public function openNewWindow(event:MouseEvent):void {
        var url:URLRequest = new URLRequest("http://mysite.com/index.jsp");
        var uv:URLVariables = new URLVariables();
        url.method = "GET";
        uv.name = "fred";
        url.data = uv;
        navigateToURL(url,"_blank");
     }
  ]]></mx:Script>
  <mx:Button label="Open New Window" click="openNewWindow(event)"/>
</mx:Application>
```

To use POST data with the URLRequest object, set the value of the URLRequest object's `method` property to POST.

Opening multiple windows with the navigateToURL() method

You can open any number of new browser windows with calls to the `navigateToURL()` method. However, because ActionScript is asynchronous, if you tried a simple loop over the method, Flash Player would only open the browser window for the last call. To avoid this, you use the `callLater()` method. This method instructs Flash Player to open a new browser window on each new frame in your application. The following example uses the `callLater()` method to open multiple browser windows with the `navigateToURL()` method:

```
<?xml version="1.0"?>
<!-- wrapper/NavigateToMultipleURLS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="openWindows(0)">
  <mx:Script><![CDATA[
     import flash.net.navigateToURL;
     private var listingData:Array=[
        "http://www.google.com",
        "http://www.adobe.com",
        "http://www.usatoday.com"
     ];

     private function openWindows(n: Number):void {
        if (n < listingData.length) {
           navigateToURL(new URLRequest(listingData[n]), '_blank');
           callLater(callLater, [openWindows, [n+1]]);
        }
     }
  ]]></mx:Script>
</mx:Application>
```

Calling JavaScript functions with the URLRequest object

You can use the URLRequest to call a JavaScript function by embedding that function in the first parameter of the method, as the following example shows:

```
public var u:URLRequest = new URLRequest("javascript:window.close()");
```

The previous code does not work on all browsers. You should include code that detects the browser type, and closes the window based on the type of browser. Also, some browsers, such as Internet Explorer, behave in unexpected ways when you invoke a URLRequest that contains JavaScript. Here are some examples:

- Executes the JavaScript URLs asynchronously. This means that it is possible to have multiple calls to the `navigateToURL()` method that is trying to execute JavaScript methods, and have only the last one occur. Each one overwrites the next.

- Stops all other navigation, such as loading of images and IFRAMEs when you call a JavaScript URL.
- Displays a security warning dialog box if the URL contains ampersand (&) or question mark (?) characters.

Invoking JavaScript with the navigateToURL() method

You can use the `navigateToURL()` method to invoke JavaScript functions on the HTML page in which the Flex application runs. However, when passing parameters by using the `navigateToURL()` method, you must enclose each parameter in quotation marks, as the following example shows:

```
<?xml version="1.0"?>
<!-- wrapper/CallingJavaScriptWithNavigateToURL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
     import flash.net.*;
     public function callWrapper(e:Event):void {
        var eType:String = String(e.type);
        var eName:String = String(e.currentTarget.id);
       var u:URLRequest = new URLRequest("javascript:catchClick('" + eName
+ "','" + eType + "')");
        navigateToURL(u,"_self");
     }
  </mx:Script>
  <mx:Button id="b1" click="callWrapper(event)" label="Call Wrapper"/>
</mx:Application>
```

The enclosing HTML wrapper includes the JavaScript to process this call, as the following example shows:

```
<html><head>
<title>wrapper/NavigateToURLWrapper.html</title>
</head>
<body scroll='no'>

<SCRIPT LANGUAGE="JavaScript">
function catchClick(name, type) {
alert(name + " triggered the " + type + " event.");
}
</SCRIPT>

<h1>Navigate To URL Wrapper</h1>

<table width='100%' height='100%' cellspacing='0' cellpadding='0'>
<tr>
<td valign='top'>
<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='200' width='400'>
<param name='src' value='CallingJavaScriptWithNavigateToURL.swf'/>
<param name='flashVars' value=''/>
<embed name='mySwf' src='CallingJavaScriptWithNavigateToURL.swf'
pluginspage='http://www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars=''/>
</object>
</td>
</tr>
</table>

</body></html>
```

You can also use the `navigateToURL()` method with basic JavaScript functions in the URLRequest itself. For example, you can launch the default e-mail client with a single line of code:

```
<?xml version="1.0"?>
<!-- wrapper/EmailWithNavigateToURL.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
     import flash.net.*;
     public function sendMail(e:Event):void {
        var u:URLRequest = new URLRequest("mailto:" + ti1.text);
        navigateToURL(u,"_self");
     }
  </mx:Script>
  <mx:Button id="b1" click="sendMail(event)" label="Send Mail"/>
  <mx:Form>
     <mx:FormItem>
        <mx:Label text="Email Address: "/>
     </mx:FormItem>
     <mx:FormItem>
        <mx:TextInput id="ti1"/>
     </mx:FormItem>
  </mx:Form>
</mx:Application>
```

Not all browsers support invoking the `javascript` protocol with the `navigateToURL()` method. If possible, you should use the `call()` method of the ExternalInterface API to invoke JavaScript methods in the enclosing HTML page. For more information, see "Using the ExternalInterface API to access JavaScript from Flex" on page 1192.

Accessing Flex from JavaScript

You can call Flex methods from your enclosing wrapper by using the ExternalInterface API. You do this by adding a public method in your Flex application to a list of callable methods. In your Flex application, you add a local Flex function to the list by using the `addCallback()` method of the ExternalInterface API. This method registers an ActionScript method as callable from the JavaScript or VBScript in the wrapper.

NOTE This feature requires that the client is running certain browsers. For more information, see "About the ExternalInterface API" on page 1184.

The signature for the `addCallback()` method is as follows:

```
addCallback(function_name:String, closure:Function):void
```

The *function_name* parameter is the name by which you call the Flex function from your HTML page's scripts. The *closure* parameter is the local name of the function that you want to call. This parameter can be a method on the application or an object instance.

The following example declares the `myFunc()` function to be callable by the wrapper:

```
<?xml version="1.0"?>
<!-- wrapper/AddCallbackExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script>
     import flash.external.*;

     public function initApp():void {
        ExternalInterface.addCallback("myFlexFunction",myFunc);
     }

     public function myFunc(s:String):void {
        l1.text = s;
     }

  </mx:Script>

  <mx:Label id="l1"/>

</mx:Application>
```

To call the Flex function from the HTML page, you get a reference to the movie object. This is the same value as the `id` and `name` properties of the `<object>` and `<embed>` tags. In this case, it is mySwf. You then call the method on that object, passing whatever parameters you want, as the following example shows:

```
<html><head>
<title>wrapper/AddCallbackWrapper.html</title>
</head>
<body scroll='no'>

<SCRIPT LANGUAGE="JavaScript">
function callApp() {
window.document.title = document.getElementById("newTitle").value;
mySwf.myFlexFunction(window.document.title);
}
</SCRIPT>

<h1>AddCallback Wrapper</h1>

<form id="f1">
Enter a new title: <input type="text" size="30" id="newTitle"
onchange="callApp()">
</form>

<table width='100%' height='100%' cellspacing='0' cellpadding='0'>
<tr>
<td valign='top'>
<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='200' width='400'>
<param name='src' value='AddCallbackExample.swf'/>
<param name='flashVars' value=''/>
<embed name='mySwf' src='AddCallbackExample.swf' pluginspage='http://
www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars=''/>
</object>
</td>
</tr>
</table>

</body></html>
```

The default value of the `id` and `name` properties in the wrapper that is generated by Flex Data Services is *mxml_filename*.mxml.swf. The name that you use to access the Flex application in your HTML page's script cannot contain any periods, so you must change the default values in the wrapper. For more information, see "Editing the Flex application's id and name properties" on page 1210.

If you do not know which browser your users will be using when they request your Flex application, you should make your wrapper's script browser independent. For more information, see "Handling multiple browser types" on page 1208.

If there is no function with the appropriate name in the Flex application or that function hasn't been made callable, the browser throws a JavaScript error.

Flex and Flash Player have strict security in place to prevent cross-site scripting. By default, Flex functions are not callable by HTML scripts. You must explicitly identify them as callable. You also cannot call a Flex function from an HTML page if the HTML page is not in the same domain as the application. However, it is possible to expand the sources from which Flex functions are called. For more information, see "About the addCallback() method" on page 1212.

Handling multiple browser types

In most cases, you must write your HTML page's scripts to handle multiple browser types. Getting a reference to the Flash application object is not the same in Internet Explorer as it is in Netscape-based browsers such as FireFox.

To write script that is browser independent, you examine the name of the navigator object and return a reference to the Macromedia Flash application based on that. The following code gets a reference to the Flash application in all major browsers:

```
<html><head>
<title>wrapper/BrowserAwareAddCallbackWrapper.html</title>
</head>
<body scroll='no'>

<SCRIPT LANGUAGE="JavaScript">
// Internet Explorer and Mozilla-based browsers refer to the Flash
application
// object differently.
// This function returns the appropriate reference, depending on the
browser.
function getMyApp(appName) {
if (navigator.appName.indexOf ("Microsoft") !=-1) {
return window[appName];
} else {
return document[appName];
}
}

function callApp() {
window.document.title = document.getElementById("newTitle").value;
getMyApp("mySwf").myFlexFunction(window.document.title);
}
</SCRIPT>

<h1>AddCallBack Wrapper</h1>

<form id="f1">
Enter a new title: <input type="text" size="30" id="newTitle"
onchange="callApp()">
</form>

<table width='100%' height='100%' cellspacing='0' cellpadding='0'>
<tr>
<td valign='top'>
<object id='mySwf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=9,0,0,0' height='200' width='400'>
<param name='src' value='AddCallbackExample.swf'/>
<param name='flashVars' value=''/>
<embed name='mySwf' src='AddCallbackExample.swf' pluginspage='http://
www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'
flashVars=''/>
</object>
</td>
```

```
</tr>
</table>

</body></html>
```

Editing the Flex application's id and name properties

By default, the name of your Flex application's Flash application object is *mxml_filename*.mxml.swf in the wrapper. However, periods are not allowed in JavaScript or VBScript method or object names. So, in order to use the ExternalInterface API, you must edit the `id` property that is assigned to your Flex application in the wrapper.

You rename the object on the page by changing the values of the `<embed>` tag's `name` property and the `<object>` tag's `id` property.

The following example shows the locations in the wrapper where you edit the application's name:

```
<noscript>
  // Set the id of the Flash application in the <object> tag.
  <object
    classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
    codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
      swflash.cab#version=9,0,0,0'
    width='300'
    height='100'
    id='MyApp'>
    <param name='flashvars' value=''>
    <param name='src' value='MyApp.mxml.swf'>

    // Set the name of the Flash application in the <embed> tag.
    <embed
      pluginspage='http://www.macromedia.com/go/getflashplayer'
      width='300'
      height='100'
      flashvars=''
      src='MyApp.mxml.swf'
      name='MyApp'
    />
  </object>
</noscript>

<script language='javascript' charset='utf-8'>
  // Change the id and name properties in the script block, too.
  ...
</script>
```

This example shows the `<object>` and `<embed>` tags in the `<noscript>` block of the wrapper. You must also change the `name` and `id` properties of the `<object>` and `<embed>` tags in the `<script>` block. These nearly identical blocks are both required to run a full-featured Flex application.

After you change the name of the Flex application object, you can reference the application by name in your JavaScript.

Handling browsers that disable JavaScript

In some cases, the client's browser either does not support JavaScript or the user has purposely disabled it. You can use the `<noscript>` tag in the wrapper to define what happens when this user tries to run your Flex applications.

One possible solution is to insert a message that tells the user that they cannot use some functionality of your application because JavaScript is disabled. The following example warns users when someone with JavaScript disabled tries to run your Flex application:

```
<html>
  <body>
    <noscript>
      <EM>Your browser either does not support JavaScript or has disabled
        it. Some features of this application require JavaScript. Click
        <A HREF="...">here</A> to continue anyway, Otherwise, please
        enable JavaScript and reload this page.
      </EM>
    </noscript>
    <script src="mysource.js"></script>
  </body>
</html>
```

About ExternalInterface API security in Flex

Allowing Flex applications to call embedded scripts on HTML pages and vice versa is subject to stringent security constraints. By default, scripts on the HTML page can communicate only with ActionScript in a Flex application if the page and the application are in the same domain. You can expand this restriction to include applications outside of the domain.

About the call() method

The success of the `call()` method depends on the HTML page's use of the `allowScriptAccess` parameter. This parameter is not an ActionScript mechanism; it is an HTML parameter. Its value determines whether your Flex application can call JavaScript in the HTML page, and it applies to all functions on the page. The default value of `allowScriptAccess` only allows communication if the Flex application and the HTML page are in the same domain.

You set the `allowScriptAccess` property of the `<object>` and `<embed>` tags on the HTML page. On the `<object>` tag, set the property as follows:

```
<object id='SendComplexDataTypes' classid='clsid:D27CDB6E-AE6D-11cf-96B8-
  444553540000' codebase='http://download.macromedia.com/pub/shockwave/
  cabs/flash/swflash.cab#version=9,0,0,0' allowScriptAccess='always'
  height='100%' width='100%'>
```

On the `<embed>` tag, set the property as follows:

```
<embed name='SendComplexDataTypes.mxml.swf' pluginspage='http://
  www.macromedia.com/go/getflashplayer'
  src='SendComplexDataTypes.mxml.swf' allowScriptAccess='always'
  height='100%' width='100%' flashvars=''/>
```

The following table describes the valid values of the `allowScriptAccess` parameter:

Value	Description
`never`	The `call()` method fails.
`sameDomain`	The `call()` method succeeds if the calling application is from same domain as the HTML page. This is the default value.
`always`	The `call()` method succeeds, regardless of whether the calling application is in the same domain as the HTML page.

About the addCallback() method

Flex prevents JavaScript methods from calling just any method in your application by requiring that you explicitly make the method callable. The default for all methods is to not be callable from JavaScript. The ExternalInterface API enables a SWF file to expose a specific interface that JavaScript can call.

By default, an HTML page can only communicate with the ActionScript in your Flex application if it originates from the same domain. You allow HTML pages outside of the Flex application's domain to call methods of your application using the `allowDomain()` method. For more information, see *Adobe Flex 2 Language Reference*.

CHAPTER 35

Using Shared Objects

35

You use the SharedObject class to store small amounts of data on the client machine. This topic describes how to use shared objects with Adobe Flex 2.

Contents

About shared objects

Shared objects function like browser cookies. You use the SharedObject class to store data on the user's local hard disk and call that data during the same session or in a later session. Applications can access only their own SharedObject data, and only if they are running on the same domain. The data is not sent to the server and is not accessible by other Flex applications running on other domains, but can be made accessible by applications from the same domain.

Shared objects compared with cookies

Cookies and shared objects are very similar. Because most web programmers are familiar with how cookies work, it might be useful to compare cookies and local SharedObjects.

Cookies that adhere to the RFC 2109 standard generally have the following properties:

- They can expire, and often do at the end of a session by default.
- They can be disabled by the client on a site-specific basis.
- There is a limit of 300 cookies total, and 20 cookies maximum per site.
- They are usually limited to a size of 4 KB each.
- They are sometimes perceived to be a security threat, and as a result, they are sometimes disabled on the client.

- They are stored in a location specified by the client browser.
- They are transmitted from client to server through HTTP.

In contrast, shared objects have the following properties:

- They do not expire by default.
- By default, they are limited to a size of 100 KB each.
- They can store simple data types (such as String, Array, and Date).
- They are stored in a location specified by the application (within the user's home directory).
- They are never transmitted between the client and server.

About the SharedObject class

Using the SharedObject class, you can create and delete shared objects, as well as detect the current size of a SharedObject object that you are using. The SharedObject class consists of the following methods:

Method	Description
`clear()`	Purges all of the data from the SharedObject object, and deletes the SharedObject file from the disk.
`flush()`	Immediately writes the SharedObject file to a file on the client.
`getLocal()`	Returns a reference to the client's domain-specific, local SharedObject object. If none exists, this method creates a new shared object on the client.
`getSize()`	Gets the size of the SharedObject file, in bytes. The default size limit is 100 KB, although it can be larger if the client allows it.

In addition to these methods, SharedObject objects have the following properties:

Property	Description
`data`	Read-only property that represents the collection of attributes the shared object stores.
`onStatus`	The shared object's event handler that is invoked for every warning, error, or informational note.

Creating a shared object

To create a SharedObject object, use the `SharedObject.getLocal()` method, which has the following syntax:

```
SharedObject.getLocal("objectName" [, pathname]): SharedObject
```

The following example creates a shared object called mySO:

```
public var mySO:SharedObject;
mySO = SharedObject.getLocal("preferences");
```

This creates a file on the client's machine called preferences.sol.

The term *local* refers to the location of the shared object. In this case, Adobe Flash Player stores the SharedObject file locally in the client's home directory.

When you create a shared object, Flash Player creates a new directory for the application and domain. It also creates an empty *.sol file that stores the SharedObject data. The default location of this file is a subdirectory of the user's home directory. On Windows, this directory is the following by default:

```
c:/Documents and Settings/username/user_domain/Application Data/Macromedia/
  Flash Player/web_domain/path_to_application/ApplicationName/
  objectName.sol
```

If you request an application named MyApp.mxml on the local host, in the Flex context, and within a subdirectory named /sos, Flash Player stores the *.sol file in the following location on Windows:

```
c:/Documents and Settings/username/user_domain/Application Data/Macromedia/
  Flash Player/#localhost/flex/sos/MyApp.mxml.swf/objectName.sol
```

NOTE If you do not provide a name in the `SharedObject.getLocal()` method, Flash Player names the file undefined.sol.

Although usually predictable, the location of the SharedObject file can be anywhere that Flash Player has access to within its sandbox and can have any name that Flash Player assigns to it.

By default, Flash can save locally persistent SharedObject objects of up to 100 KB per domain. When the application tries to save data to a shared object that would make it bigger than 100 KB, Flash Player displays the Local Storage dialog box, which lets the user allow or deny local storage for the domain that is requesting access.

Specifying a path

You can use the optional *pathname* parameter to specify a location for the SharedObject file. This file must be a subdirectory of that domain's SharedObject directory. For example, if you request an application on the localhost and specify the following:

```
mySO = SharedObject.getLocal("myObjectFile","/");
```

Flash Player writes the SharedObject file in the /#localhost directory. This is useful if you want more than one application on the client to be able to access the same shared object. In this case, the client could run two Flex applications, both of which specify a path to the shared object that is the root of the domain; the client could then access the same shared object from both applications. To share data between more than application without persistence, you can use the LocalConnection object.

If you specify a directory that does not exist, Flash Player does not create a SharedObject file.

Adding data to a shared object

You add data to a SharedObject's *.sol file using the `data` property of the SharedObject object. To add new data to the shared object, use the following syntax:

```
sharedObject_name.data.variable = value;
```

The following example adds the `userName`, `itemNumbers`, and `adminPrivileges` properties and their values to a SharedObject:

```
public var currentUserName:String = "Reiner";
public var itemsArray:Array = new Array(101,346,483);
public var currentUserIsAdmin:Boolean = true;
mySO.data.userName = currentUserName;
mySO.data.itemNumbers = itemsArray;
mySO.data.adminPrivileges = currentUserIsAdmin;
```

After you assign values to the `data` property, you must instruct Flash Player to write those values to the SharedObject's file. To force Flash Player to write the values to the SharedObject's file, use the `SharedObject.flush()` method, as follows:

```
mySO.flush();
```

If you do not call the `SharedObject.flush()` method, Flash Player writes the values to the file when the application quits. However, this does not provide the user with an opportunity to increase the available space that Flash Player has to store the data if that data exceeds the default settings. Therefore, it is a good practice to call `SharedObject.flush()`.

You can store typed ActionScript instances in shared objects. You do this by calling the `flash.net.registerClassAlias()` method to register the class. If you create an instance of your class and store it in the data member of your shared object and later read the object out, you will get a typed instance. By default, the SharedObject `objectEncoding` property supports AMF3 encoding, and unpacks your stored instance from the SharedObject object; the stored instance retains the same type you specified when you called the `registerClassAlias()` method.

Creating multiple shared objects

You can create multiple shared objects for the same Flex application. To do this, you assign each of them a different instance name, as the following example shows:

```
public var mySO:SharedObject = SharedObject.getLocal("preferences");
public var mySO2:SharedObject = SharedObject.getLocal("history");
```

This creates a preferences.sol file and a history.sol file in the Flex application's local directory.

Destroying shared objects

To destroy a SharedObject on the client, use the `SharedObject.clear()` method. This does not destroy directories in the default path for the application's shared objects.

The following example deletes the SharedObject file from the client:

```
public function destroySharedObject():void {
  mySO.clear();
}
```

SharedObject example

The following example shows that you can store simple objects, such as a Date object, in a SharedObject object without having to manually serialize and deserialize those objects.

The following example begins by welcoming you as a first-time visitor. When you click Log Out, the application stores the current date in a shared object. The next time you launch this application or refresh the page, the application welcomes you back with a reminder of the time you logged out.

To see the application in action, launch the application, click Log Out, and then refresh the page. The application displays the date and time that you clicked the Log Out button on your previous visit. At any time, you can delete the stored information by clicking the Delete LSO button.

```
<?xml version="1.0"?>
<!-- lsos/WelcomeMessage.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initApp()">
  <mx:Script><![CDATA[
  public var mySO:SharedObject;
  [Bindable]
  public var welcomeMessage:String;

  public function initApp():void {
     mySO = SharedObject.getLocal("mydata");
     if (mySO.data.visitDate==null) {
        welcomeMessage = "Hello first-timer!"
     } else {
        welcomeMessage = "Welcome back. You last visited on " +
           getVisitDate();
     }
  }

  private function getVisitDate():Date {
     return mySO.data.visitDate;
  }

  private function storeDate():void {
     mySO.data.visitDate = new Date();
     mySO.flush();
  }

  private function deleteLSO():void {
     // Deletes the SharedObject from the client machine.
     // Next time they log in, they will be a 'first-timer'.
     mySO.clear();
  }

  ]]></mx:Script>
  <mx:Label id="label1" text="{welcomeMessage}"/>
  <mx:Button label="Log Out" click="storeDate()"/>
  <mx:Button label="Delete LSO" click="deleteLSO()"/>
</mx:Application>
```

For more examples of using shared objects, see the Flex example applications in the samples.war file.

CHAPTER 36

Creating Accessible Applications

36

Adobe Flex 2 provides accessibility features that let you create applications that are accessible to users with disabilities. As you design accessible applications, consider how your users will interact with the content. Visually impaired users, for example, might rely on assistive technology such as screen readers, which provide an audio version of screen content, or screen magnifiers, which display a small portion of the screen at a larger size, effectively reducing the visible screen area. Hearing-impaired users might read text and captions in the document in place of audio content. Other considerations arise for users with mobility or cognitive impairments.

This topic describes the accessibility features of Flex.

Contents

Accessibility overview

You create accessible content by using accessibility features included with Flex, by taking advantage of ActionScript designed to implement accessibility, and by following recommended design and development practices. The following list of recommended practices is not exhaustive, but suggests common issues to consider. Depending on your audience's needs, additional requirements may arise.

Visually impaired users For visually impaired users, keep in mind the following design recommendations:

- Design and implement a logical tab order for the tabs.
- Design the document so that constant changes in content do not unnecessarily cause screen readers to refresh. For example, you should group or hide looping elements.
- Provide captions for narrative audio. Be aware of audio in your document that might interfere with a user being able to listen to the screen reader.
- Use percentage sizing so that your applications scale properly at smaller screen sizes. This allows users of screen magnifiers to see more of your application at one time. Also take into account that many visually impaired users run applications with lower screen resolutions than other users.
- Ensure that foreground and background colors contrast sufficiently to make text readable for people with low vision.
- Ensure that controls don't depend on the use of a specific pointer device, such as a mouse or trackball.
- Ensure that components are accessible by keyboard. All Flex components defined as accessible include keyboard navigation. For a list of these components and the available keyboard commands for each, see "Accessible components and containers" on page 1224.

Color blind users For color blind users, ensure that color is not the only means of conveying information.

Users with mobility impairment For users with mobility impairment, keep in mind the following design recommendations:

- Ensure that controls don't depend on the use of a specific pointer device.
- Ensure that components are accessible by keyboard. All Flex components defined as accessible include keyboard navigation. For a list of these components and the available keyboard commands for each, see "Accessible components and containers" on page 1224.

Hearing-impaired users For hearing-impaired users, ensure that you add captions to audio content.

Users with cognitive impairment For users with cognitive impairments, such as dyslexia, keep in mind the following design recommendations:

- Ensure an uncluttered, easy-to-navigate design.
- Provide graphical imagery that helps convey the purpose and message of the application. These graphics should enhance, not replace, textual or audio content.
- Provide more than one method to accomplish common tasks.

About worldwide accessibility standards

Many countries, including the United States, Australia, Canada, Japan, and countries in the European Union, have adopted accessibility standards based on those developed by the World Wide Web Consortium (W3C). The W3C publishes *Web Content Accessibility Guidelines*, a document that prioritizes actions that designers should take to make web content accessible. For information about the Web Accessibility Initiative, see the W3C website at www.w3.org/WAI.

In the United States, the law that governs accessibility is commonly known as Section 508, which is an amendment to the U.S. Rehabilitation Act. Section 508 prohibits federal agencies from buying, developing, maintaining, or using electronic technology that is not accessible to those with disabilities. In addition to mandating standards, Section 508 lets government employees and the public sue agencies in federal court for noncompliance.

For additional information about Section 508, see the U.S. government-sponsored website at www.section508.gov.

Viewing the Flex Accessibility web page

This topic contains an introduction to the accessibility features in Flex and to developing accessible applications. For the latest information on creating and viewing accessible Flex content, including supported platforms, known issues, screen reader compatibility, articles, and accessible examples, see the Flex Accessibility web page at www.adobe.com/go/flex_accessibility.

About screen reader technology

A screen reader is software designed to navigate through a website and read the web content aloud. Visually impaired users often rely on this technology. You can create content designed for use with screen readers for Microsoft® Windows® platforms only. Users who view your content must have Adobe Flash Player 9 or later, and Internet Explorer on Windows 2000 or Windows XP or later.

JAWS, from Freedom Scientific, is one example of screen reader software. You can access the JAWS page on the Freedom Scientific website at www.hj.com/fs_products/software_jaws.asp. Another commonly used screen reader program is Window-Eyes, from GW Micro. To access the latest information on Window-Eyes, visit the GW Micro website at www.gwmicro.com.

NOTE	Flex support is most comprehensive in the JAWS screen reader. You must have JAWS version 6.10.1006 or later.

Screen readers help users understand what is contained in a web page or Flex application. Based on the keyboard shortcuts that you define, you can let users easily navigate through your application by using the screen reader.

Because different screen reader applications use various methods to translate information into speech, your content will vary in how it's presented to each user. As you design accessible applications, keep in mind that you have no control over how a screen reader behaves. You can only mark up the content in your applications so that you expose the text and ensure that screen reader users can activate the controls. This means that you can decide which objects in the application are exposed to screen readers, provide descriptions for the objects, and decide the order in which the objects are exposed to screen readers. However, you cannot force screen readers to read specific text at specific times or control the manner in which that content is read.

To use a JAWS screen reader with a Flex application, users must download a set of scripts before invoking the Flex application. For more information, see "Configuring a JAWS screen reader for Flex applications" on page 1224.

Flash Player and Microsoft Active Accessibility

Adobe Flash Player uses Microsoft Active Accessibility (MSAA), which provides a descriptive and standardized way for applications and screen readers to communicate. MSAA is available for Windows operating systems only. For more information on Microsoft Accessibility Technology, visit the Microsoft Accessibility website at www.microsoft.com/enable/default.aspx.

The Windows ActiveX (Internet Explorer plug-in) version of Flash Player 9 supports MSAA, but the Windows Netscape and Windows stand-alone players currently do not.

Flex supports a debugger version of Flash Player that can display debugging information during run time, and generate profiling information so that you can more easily develop applications. However, the debugger version of Flash Player does not support accessibility.

CAUTION	MSAA is currently *not* supported in the opaque windowless and transparent windowless modes. (These modes are options in the HTML Publish Settings panel, available for use with the Windows version of Internet Explorer 4.0 or later, with the Flash ActiveX control.) If your Flash content must be accessible to screen readers, avoid using these modes.

Configuring Flex applications for accessibility

This section describes how to enable the accessibility features of Flex and how to configure a screen reader for use with Flex applications.

Enabling accessibility in Flex

By default, Flex accessibility features are not enabled. When you enable accessibility, you enable the application to communicate with a screen reader.

To enable accessibility, you can use one of the following methods:

- Enable accessibility by default for all Flex applications so that all requests return accessible content.

  To enable accessibility for all Flex applications, edit the flex-config.xml file to set the `<accessible>` property to `true`, as the following example shows:

  ```
  <compiler>
    ...
    <accessible>true</accessible>
    ...
  <compiler>
  ```

- Enable accessibility when you are using the mxmlc command-line compiler.

  When you compile a file by using the mxmlc command-line compiler, you can use the `-accessible` option to enable accessibility, as the following example shows:

  ```
  mxmlc -accessible c:/dev/myapps/mywar.war/app1.mxml
  ```

  For more information on the command-line compiler, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

If you are building applications for Adobe Flex Data Services, and are deploying the application as MXML files, you can enable accessibility on an individual request by setting the `accessible` query parameter to `true`, as the following example shows:

```
http://www.mycompany.com/myflexapp/app1.mxml?accessible=true
```

If you edited the flex-config.xml file to enable accessibility by default, you can disable it for an individual request by setting the `accessible` query parameter to `false`, as the following example shows:

```
http://www.mycompany.com/myflexapp/app1.mxml?accessible=false
```

For more information on the command-line compiler, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Configuring a JAWS screen reader for Flex applications

To use the JAWS screen reader with a Flex application, users must download scripts from the Adobe accessibility website before invoking a Flex application. Screen readers work best when in Forms mode, which lets users interact directly with the Flex application. These scripts let users switch between Virtual Cursor mode and Forms mode by using the Enter key from almost anywhere within a Flex application. If necessary, users can exit Forms mode by using the standard JAWS keystrokes.

Users can download these scripts, and the installation instructions, from the Adobe website at www.adobe.com/go/flex_accessibility.

To verify that the Flex scripts for JAWS are correctly installed, users can press the Insert+Q keys when JAWS is running. If the scripts are installed correctly, users hear "Updated for Adobe Flex 1.5 and 2" in the voiced response to this key stroke.

It is important that you direct users with visual impairments to the script download page so that they have the necessary scripts to use JAWS effectively.

Accessible components and containers

To accelerate building accessible applications, Adobe built support for accessibility into Flex components and containers. These components and containers automate many of the most common accessibility practices related to labeling, keyboard access, and testing. They also help ensure a consistent user experience across rich Internet applications.

Flex comes with the following set of accessible components and containers:

Component	Screen reader behavior
Accordion container	Press the arrow keys to move the focus to a different panel, and then use the Spacebar or Enter key to select that panel. Use the Page Up and Page Down keys to move between individual panels of the container. When a screen reader encounters an Accordion container, it indicates each panel with the word *tab*. It indicates the current pane with the word *active*. For more information on keyboard navigation, see "Accordion container Keyboard navigation" on page 642.
Alert control	In Forms mode, the text in the Alert control is announced, and the label of its default button. When not in Forms mode, the text in the Alert control is announced twice when you press the Down Arrow.

Component	Screen reader behavior
Button control	Press the Spacebar to activate the Button control. To cancel activating a button, press the Tab key to move the focus off the Button control before releasing the Spacebar. When a screen reader encounters a Button control, activation varies, depending on the screen reader. In JAWS 6.10, the Spacebar activates Button controls when Forms mode is active. When Forms mode is inactive, the Spacebar or Enter key can be used to activate Button controls. For more information on keyboard navigation, see "Button control user interaction" on page 271.
CheckBox control	Press the Spacebar to activate the check box items. For more information on keyboard navigation, see "CheckBox control user interaction" on page 288.
ComboBox control	For more information on keyboard navigation, see "ComboBox control user interaction" on page 465.
DataGrid control	Press the arrow keys to highlight the contents, and then move between the individual characters within that field. When using a screen reader in forms mode, use the Tab key to move between editable TextInput fields in the DataGrid control. For more information on keyboard navigation, see "DataGrid control user interaction" on page 478.
DateChooser control	Press the Up, Down, Left, and Right Arrow keys to change the selected date. Use the Home key to reach the first enabled date in the month and the End key to reach the last enabled date in a month. Use the Page Up and Page Down keys to reach the previous and next months. For more information on keyboard navigation, see "User interaction" on page 306.
DateField control	Use the Control+Down Arrow keys to open the DateChooser control and select the appropriate date. When using a screen reader in Forms mode, use the same keystrokes as for keyboard navigation. When a screen reader encounters a DateChooser control in Forms mode, it announces the control as "DropDown Calendar *currentDate*, to open press ControlDown, ComboBox", where *currentDate* is the currently selected date. For more information on keyboard navigation, see "User interaction" on page 306.
Form container	For information on keyboard navigation, see "Defining a default button" on page 578.

Component	Screen reader behavior
Image control	An Image control with a ToolTip defined is read by a screen reader only when Forms mode is inactive. The Image control is not focusable in Forms mode, or by the keyboard.
Label control	A Label control is read by a screen reader when it is associated with other controls, or when the Forms mode is inactive. The Label control is not focusable in Forms mode, or by the keyboard.
LinkButton control	LinkButton control activation when using a screen reader varies, depending on the screen reader. In JAWS 6.10, press the Spacebar to activate a LinkButton control when in Forms mode. When Forms mode is inactive, use the Spacebar or Enter key to activate the control. For more information on keyboard navigation, see “LinkButton control user interaction” on page 310.
List control	Screen reader navigation is the same as for keyboard navigation. For more information on keyboard navigation, see “Keyboard navigation” on page 453. For information on creating data tips (tooltips for individual list elements), see “Displaying DataTips” on page 444. For information on creating scroll tips (tooltips that provide information while the user scrolls through a list), see “Displaying ScrollTips” on page 444.
Menu control	Screen reader navigation is the same as for keyboard navigation. For more information on keyboard navigation, see “Menu control user interaction” on page 430.
MenuBar control	Screen reader navigation is the same as for keyboard navigation. For more information on keyboard navigation, see “Menu control user interaction” on page 430.
Panel container	Screen reader announces the panel title only when Forms mode is inactive.
RadioButton control	With one radio button selected within a group, press the Enter key to enter that group. Use the arrow keys to move between items in that group. Press the Down and Right Arrow keys to move to the next item in a group; press the Up and Left Arrow keys to move to a previous item in the group. When using a screen reader, select a radio button by using the Spacebar key. For more information on keyboard navigation, see “RadioButton user interaction” on page 291.
RadioButtonGroup control	Screen reader navigation is the same as for the RadioButton control.

Component	Screen reader behavior
TabNavigator container	When a screen reader encounters a TabNavigator container pane, it indicates each pane with the word *tab*. It indicates the current pane with the word *active*. When a pane is selected, the user moves to that panel by pressing the Enter key. Press the arrow keys to move the focus to a different panel, and then use the Spacebar or Enter key to select that panel. Use the Page Up and Page Down keys to move between individual panels of the container. For more information on keyboard navigation, see "TabNavigator container Keyboard navigation" on page 638.
Text control	A Text control is not focusable, and is only read by screen readers when Forms mode is inactive.
TextArea control	Use the Home or Page Down key to move to the beginning of a line. Use the End or Page Up key to move to the end of a line.
TextInput control	Use the Home or Page Down key to move to the beginning of a line. Use the End or Page Up key to move to the end of a line.
TitleWindow container	A screen reader announces the TitleWindow control only when Forms mode is inactive.
ToolTipManager	When a screen reader is used, the contents of a tooltip are read after the item to which the tooltip is attached gets focus. Tooltips attached to nonaccessible components (other than the Image control) are not read.
Tree control	Press the Up and Down Arrow keys to move between items in a Tree control. To open a group, press the Right Arrow key or Spacebar. To close a group, press the Left Arrow key or Spacebar. For more information on keyboard navigation, see "Editing a node label at run time" on page 488 and "Tree user interaction" on page 488.

Creating tab order and reading order

There are two aspects of tab indexing order—the *tab order* in which a user navigates through the web content, and the *reading order* in which things are read by the screen reader.

Flash Player uses a tab index order from left to right and top to bottom. However, if this is not the order you want to use, you can customize both the tab order and the reading order by using the `InteractiveObject.tabIndex` property. (In ActionScript, the `tabIndex` property is synonymous with the reading order.)

Tab order You can use the `tabIndex` property of every component to create a tab order that determines the order in which objects receive input focus when a user presses the Tab key.

Reading order You can use the `tabIndex` property to control the order in which the screen reader reads information about the object. To create a reading order, you assign a value to the `tabIndex` property for every component in your application. You should set the `tabIndex` property for every accessible object, not just the focusable objects. For example, you must set the `tabIndex` property for a Text control even though a user cannot tab to it. If you do not set the `tabIndex` property for every accessible object, Flash Player puts that object at the end of the tab order, rather than in its appropriate tab order location.

Scrolling to a component when tabbing

As a general rule, you should structure your application so that all components fit in the available screen area, otherwise Flex adds vertical or horizontal scroll bars as necessary. Scroll bars let users move around the application to access components outside of the screen area.

If your application uses scroll bars, tabbing through the application components does not automatically scroll the application to make the currently selected component visible. You can add logic to your application to automatically scroll to the currently selected component, as the following example shows:

```
<?xml version="1.0"?>
<!-- accessibility\ScrollComp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="setupFocusViewportWatcher();">

    <mx:Script>
        <![CDATA[

        import mx.core.Container;
        import mx.core.EdgeMetrics;

        [Bindable]
        public var cards: Array = [
            {label:"Visa", data:1},
            {label:"Master Card", data:2},
            {label:"American Express", data:3} ];

        [Bindable]
        public var selectedItem:Object;

        [Bindable]
        public var forListDP:Array = [
            {label:'Apple', data:10.00},
            {label:'Banana', data:15.00},
            {label:'Melon', data:3.50},
            {label:'Kiwi', data:7.65},
            {label:'123', data:12.35 },
            {label:'some', data:10.01 }];

        // Set up the event listener for the focusIn event.
        public function setupFocusViewportWatcher():void {
            addEventListener("focusIn", makeFocusedItemVisible);
        }

        public function makeFocusedItemVisible(event:FocusEvent):void {
            // Target is the actual object that has focus.
            var target:InteractiveObject = InteractiveObject(event.target);

            // OriginalTarget is the component that has focus as some
           // component actually delegate true focus to an internal object.
            var originalTarget:InteractiveObject =
        InteractiveObject(focusManager.findFocusManagerComponent(target));

            // The viewable portion of a container
```

```
var viewport:Rectangle = new Rectangle();
do {
    // Cycle through all parents looking for containers.
    if (target.parent is Container) {
        var viewportChanged:Boolean = false;
        var c:Container = target.parent as Container;

        // Get the viewable area in the container.
        var vm:EdgeMetrics = c.viewMetrics;
        viewport.x = vm.left;
        viewport.y = vm.top;
        viewport.width =
            c.width / c.scaleX - vm.left - vm.right;
        viewport.height =
            c.height / c.scaleY - vm.top - vm.bottom;

       // Calculate the position of the target in the container.
        var topLeft:Point = new Point(0, 0);
        var bottomRight:Point =
    new Point(originalTarget.width, originalTarget.height);
        topLeft = originalTarget.localToGlobal(topLeft);
        topLeft = c.globalToLocal(topLeft);
       bottomRight = originalTarget.localToGlobal(bottomRight);
        bottomRight = c.globalToLocal(bottomRight);

        // Figure out if we have to move the scroll bars.
       // If the scroll bar moves, the position of the component
        // moves as well. This algorithm makes sure the top
        // left of the component is visible if the component is
        // bigger than the viewport.
        var delta:Number;

        if (bottomRight.x > viewport.right) {
            delta = bottomRight.x - viewport.right;
            c.horizontalScrollPosition += delta;
            topLeft.x -= delta;
            viewportChanged = true;
        }

        if (topLeft.x < viewport.left) {
            // leave it a few pixels in from the left
            c.horizontalScrollPosition -=
                viewport.left - topLeft.x + 2;
            viewportChanged = true;
        }

        if (bottomRight.y > viewport.bottom) {
            delta = bottomRight.y - viewport.bottom;
            c.verticalScrollPosition += delta;
            topLeft.y -= delta;
```

```
                    viewportChanged = true;
                }

                if (topLeft.y < viewport.top) {
                    // leave it a few pixels down from the top
                    c.verticalScrollPosition -=
                        viewport.top - topLeft.y + 2;
                    viewportChanged = true;
                }

                // You must the validateNow() method to get the
                // container to move the component before working
                // on the next parent.
                // Otherwise, your calculations will be incorrect.
                if (viewportChanged) {
                    c.validateNow();
                }
            }

            target = target.parent;
        }

        while (target != this);
    }
    ]]>
</mx:Script>

<mx:Model id="statesModel" source="assets/states.xml"/>
<mx:Panel
        x="58" y="48"
        width="442" height="201"
        layout="absolute"
        title="Tab through controls to see if focus stays in view">

    <mx:VBox x="10" y="10" verticalScrollPolicy="off">
        <mx:TextInput/>
        <mx:TextInput/>
        <mx:TextArea width="328" height="64"/>
        <mx:ComboBox dataProvider="{cards}" width="150"/>
        <mx:DataGrid dataProvider="{forListDP}" />
        <mx:DateChooser yearNavigationEnabled="true"/>
        <mx:List id="source"
            width="75"
            dataProvider="{statesModel.state}"/>
    </mx:VBox>
</mx:Panel>
</mx:Application>
```

Creating accessibility with ActionScript

For accessibility properties that apply to the entire document, use the flash.accessibility.AccessibilityProperties class. For more information on this class, see *Adobe Flex 2 Language Reference*.

The following table lists some of the relevant properties of the AccessibilityProperties class:

Property	Type	Description
`description`	String	Specifies a description for the component that is read by the screen reader.
`forceSimple`	Boolean	Hides the children of a component from a screen reader when set to `true`. The default value is `false`.
`name`	String	Specifies a description of the component that is read by the screen reader. When accessible objects do not have a specified name, a screen reader uses a generic word, such as *Button*.
`shortcut`	String	Indicates a keyboard shortcut associated with this display object.
`silent`	Boolean	Hides a component from a screen reader when set to `true`. The default value is `false`.

Modifying these properties has no effect by itself. You must also use the `Accessibility.updateProperties()` method to inform screen reader users of Flash Player content changes. Calling this method causes Flash Player to re-examine all accessibility properties, update property descriptions for the screen reader, and, if necessary, send events to the screen reader that indicate changes occurred.

When updating the accessibility properties of multiple objects at once, you must include only a single call to the `Accessiblity.updateProperties()` method. (Excessive updates to the screen reader can cause some screen readers to become too verbose.)

Implementing screen reader detection with the Accessibility.isActive() method

To create content that behaves in a specific way if a screen reader is active, you can use the ActionScript `Accessibility.active` property, which is set to a value of `true` if a screen reader is present, and `false` otherwise. You can then design your content to perform in a way that is compatible with screen reader use, such as by hiding child elements from the screen reader.

For example, you could use the `Accessibility.active` property to decide whether to include unsolicited animation. *Unsolicited animation* means animation that happens without the screen reader doing anything. This can be very confusing for screen readers.

For more information on the Accessibility class, see *Adobe Flex 2 Language Reference*.

Accessibility for hearing-impaired users

To provide accessibility for hearing-impaired users, you can include captions for audio content that is integral to the comprehension of the material presented. A video of a speech, for example, would probably require captions for accessibility, but a quick sound associated with a button probably would not require a caption.

Testing accessible content

When you test your accessible applications, follow these recommendations:

- Ensure that you have enabled accessibility. For more information, see "Configuring Flex applications for accessibility" on page 1223.
- Test and verify that users can navigate your interactive content effectively by using only the keyboard. This can be an especially challenging requirement, because different screen readers work in different ways when processing input from the keyboard, which means that your content might not receive keystrokes as you intended. Ensure that you test all keyboard shortcuts.
- Ensure that graphic content has alternative representation by checking for tooltips on every image.
- Ensure that your content is resizable or sized to work with screen magnifiers. It should clearly present all material at smaller sizes.
- Ensure that any substantive audio is captioned. To test for content loss, try using your application with your sound muted or wearing headphones that pipe in unrelated music.
- Ensure that you use a clean and simple interface to help screen readers and users with cognitive impairments navigate through your application. Give your application to people who have never seen it before and watch while they try to perform key tasks without any assistance or documentation. If they fail, consider modifying your interface.
- Test your application with a screen reader, or have it tested by users of screen readers.

PART 5

Flex Data Features

5

This part describes how to use Adobe Flex 2 data representation and data features. For information on accessing data, see Part 6, "Data Access and Interconnectivity".

The following topics are included:

CHAPTER 37

Representing Data

37

This topic describes data representation, which is a combination of features that provides a powerful way to validate, format, and store data, and pass data between objects.

Contents

About data representation

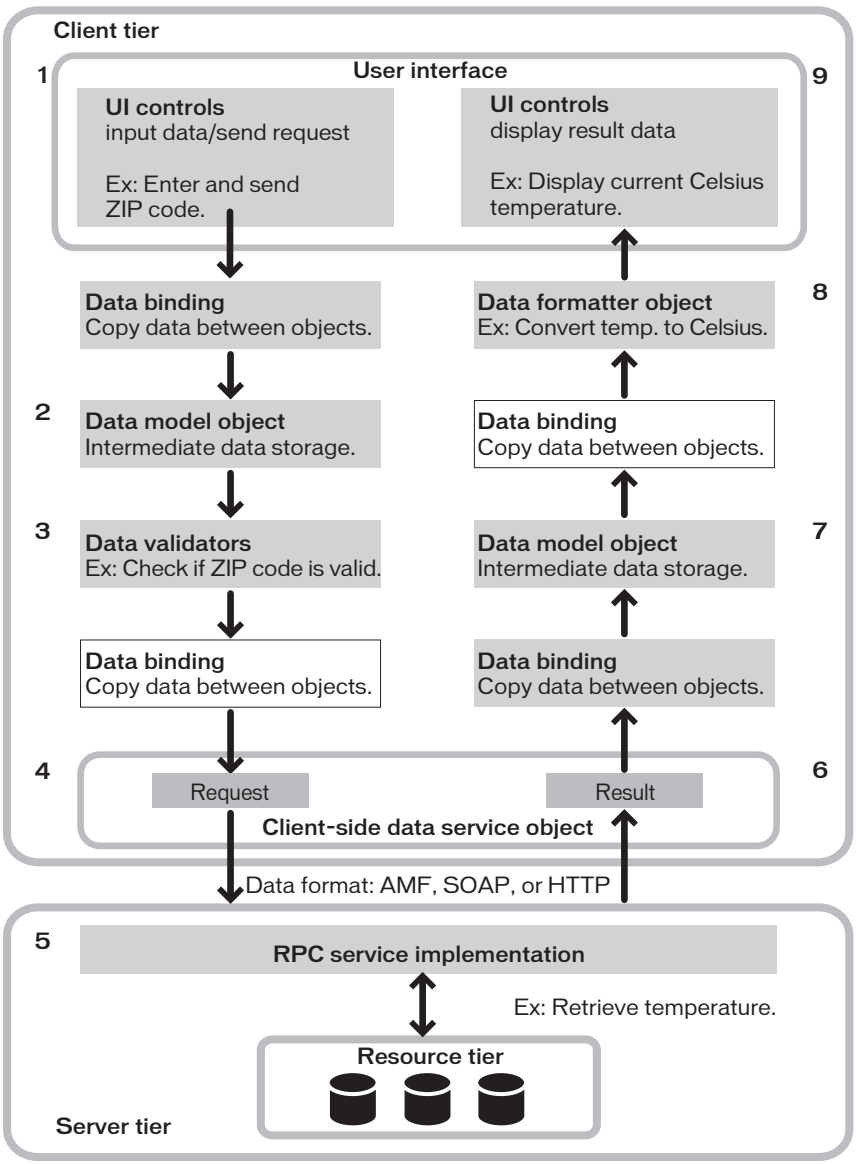
Adobe Flex 2 provides the following set of features for representing data in your applications: data binding, validation, and formatting. These features work in conjunction with the Adobe Flex Data Services features. Together, they allow you to perform the following tasks:

- Pass data between client-side objects.
- Store data in client-side objects.
- Validate data before passing it between client-side objects.
- Format data before displaying it.

The following steps describe a simple scenario in which a user provides input data and requests information in an Adobe Flex application:

1. The user enters data in input fields and submits a request by clicking a Button control.
2. (Optional) *Data binding* passes data to a data model object, which provides intermediate data storage. This allows data to be manipulated and passed to other objects in the application.
3. (Optional) One or more *data validator* objects validate the request data. Validator objects check whether the data meets specific criteria.
4. The data is passed to a server-side object.
5. The server-side object processes the request and returns data or a fault object if a valid result cannot be returned.
6. (Optional) Data binding passes data to a data model object, which provides intermediate data storage. This allows data to be manipulated and passed to other objects in the application.
7. (Optional) One or more *data formatter* objects format result data for display in the user interface.
8. Data binding passes data into user interface controls for display.

The following diagram illustrates what happens in Flex for two different user input examples. In one example, the user enters ZIP code data, and Flex validates the format. In the other example, the user requests the current temperature in Celsius.



Data binding

The data binding feature provides a syntax for automatically copying the value of a property of one client-side object to a property of another object at run time. Data binding is usually triggered when the value of the source property changes. You can use data binding to pass user input data from user interface controls to a data service. You can also use data binding to pass results returned from a data service to user interface controls.

The following example shows a Text control that gets its data from a Slider control's `value` property. The property name inside the curly braces ({ }) is a binding expression that copies the value of the source property, `mySlider.value`, into the Text control's `text` property.

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

For more information, see Chapter 38, "Binding Data," on page 1245.

Data models

The data model feature lets you store data in client-side objects. A *data model* is an ActionScript object that contains properties for storing data, and that optionally contains methods for additional functionality. Data models are useful for partitioning the user interface and data in an application.

You can use the data binding feature to bind user interface data into a data model. You can also use the data binding feature to bind data from a data service to a data model.

You can define a simple data model in an MXML tag. When you require functionality beyond storage of untyped data, you can use an ActionScript class as a data model.

The following example shows an MXML-based data model with properties of TextInput controls bound into its fields:

```
<?xml version="1.0"?>
<!-- datarep\ModelTag.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
     <mx:Model id="reg">
       <registration>
          <name>{nme.text}</name>
          <email>{email.text}</email>
          <phone>{phone.text}</phone>
          <zip>{zip.text}</zip>
          <ssn>{ssn.text}</ssn>
       </registration>
    </mx:Model>
    <mx:TextInput id="nme"/>
    <mx:TextInput id="email"/>
    <mx:TextInput id="phone"/>
    <mx:TextInput id="zip"/>
    <mx:TextInput id="ssn"/>
</mx:Application>
```

For more information about data models, see Chapter 39, “Storing Data,” on page 1269.

Data validation

The data validation feature lets you ensure that data meets specific criteria before the application uses the data. *Data validators* are ActionScript objects that check whether data in a component is formatted correctly. You can apply a data validator to a property of any component. For models in a remote procedure call (RPC) component declaration, properties to which a validator component is applied are validated just before the request is sent to an RPC service destination. Only valid requests are sent.

The following example shows MXML code that uses the standard ZipCodeValidator component, represented by the `<mx:ZipCodeValidator>` tag, to validate the format of the ZIP code that a user enters. The `source` property of the ZipCodeValidator validator indicates the property that it validates.

```
<?xml version="1.0"?>
<!-- datarep\Validate.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:TextInput id="input" text="enter zip" width="80"/>

    <mx:Model id="zipModel">
        <root>
            <zip>{input.text}</zip>
        </root>
    </mx:Model>

    <mx:ZipCodeValidator source="{zipModel}" property="zip"
        listener="{input}" trigger="{input}"/>
</mx:Application>
```

For more information about validator components, see Chapter 40, “Validating Data,” on page 1281.

Data formatting

The data formatting feature lets you change the format of data before displaying it in a user interface control. For example, when a data service returns a string that you want to display in the (xxx)xxx-xxxx phone number format, you can use a formatter component to ensure that the string is reformatted before it is displayed.

A *data formatter component* is an object that formats raw data into a customized string. You can use data formatter components with data binding to reformat data that is returned from a data service.

The following example declares a DateFormatter component with an MM/DD/YYYY date format, and binds the formatted version of a Date object returned by a web service to the `text` property of a TextInput control:

```
<?xml version="1.0"?>
<!-- datarep\Validate.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:WebService id="myService" destination="Shop"/>

    <!-- Declare a formatter and specify formatting properties. -->
    <mx:DateFormatter id="StandardDateFormat" formatString="MM/DD/YYYY"/>

    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput
        text="Your order shipped on
{StandardDateFormat.format(myService.purchase.result.date)}"/>
</mx:Application>
```

For more information about data formatters, see Chapter 41, “Formatting Data,” on page 1327.

CHAPTER 38

Binding Data

38

This topic describes data binding, which is a feature that lets you pass data between client-side objects in an application. Binding automatically copies the value of a property of a source object to a property of a destination object when the source property changes. Binding lets you pass data between the different layers of the application, such as the user interface, data models, and data services.

Contents

About data binding

Data binding is the process of tying the data in one object to another object. It provides a convenient way to pass data around in an application. Adobe Flex 2 provides three ways to specify data binding: the curly braces (`{}`) syntax and the `<mx:Binding>` tag in MXML and the BindingUtils methods in ActionScript.

Common uses of data binding include the following:

- To bind properties of user interface controls to data service requests.
- To bind data service results to properties of user interface controls.
- To bind data service results to a middle-tier data model, and to bind that data model's fields to user interface controls. For more information about data models, see Chapter 39, "Storing Data," on page 1269.
- To bind properties of user interface controls to a middle-tier data model, and to bind that data model's fields bound to a data service request (a three-tier system).
- To bind an ArrayCollection or XMLListCollection object to the `dataProvider` property of a List-based control.
- To bind individual parts of complex properties to properties of user interface controls. An example would be a master-detail scenario in which clicking an item in a List control displays data in several other controls.
- To bind XML data to user interface controls by using ECMAScript for XML (E4X) expressions in binding expressions.

Although binding is a powerful mechanism, it is not appropriate for all situations. For example, for a complex user interface in which individual pieces must be updated based on strict timing, it would be preferable to use a method that assigns properties in order. Also, binding executes every time a property changes, so it is not the best solution when you want changes to be noticed only some of the time.

Data binding requires a source property, a destination property, and a triggering event that indicates when to copy the data from the source to the destination. The following example shows a Text control that gets its data from an HSlider control's `value` property. The property name inside the curly braces is the source property of the binding expression. When the value of the source property changes, Flex copies the current value of the source property, `mySlider.value`, to the destination property, the Text control's `text` property.

```
<?xml version="1.0"?>
<!-- binding/BasicBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HSlider id="mySlider"/>
    <mx:Text text="{mySlider.value}"/>
</mx:Application>
```

You can use all properties of a component as the destination of a data binding expression. However, to use a property as the source of a data binding expression, the component must be implemented to support data binding, which means that the component dispatches an event when the value of the property changes to trigger the binding. For more information on creating component properties that can be used as the source of a data binding expression, see "Bindable metadata tag" in *Creating and Extending Flex 2 Components*.

In the *Adobe Flex 2 Language Reference*, a property that can be used as the source of a data binding expression includes the following statement in its description:

"This property can be used as the source for data binding."

Binding occurs under the following circumstances:

- The object that is the binding source dispatches an event.
- Application code calls a service.

In addition to properties, you can use ActionScript functions as the source of binding expressions. You usually do this when using a bindable property as an argument of a function. When the property changes, the function executes, and the result is used in the binding destination.

You can also use a property of type Function as a binding source or destination. A property of type Function is a variable that holds a reference to a function.

Binding data with curly braces

Using the curly braces syntax is the simplest way to pass data between objects in an application. When you use this syntax, you put curly braces ({ }) around a binding source as the value of a destination property.

In the following example, a set of control properties acts as the binding source for a data model. For more information about data models, see Chapter 39, "Storing Data," on page 1269.

```
<?xml version="1.0"?>
<!-- binding/BindingBraces.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Data model stores registration data that user enters. -->
    <mx:Model id="reg">
        <registration>
            <name>{fullname.text}</name>
            <email>{email.text}</email>
            <phone>{phone.text}</phone>
            <zip>{zip.text}</zip>
            <ssn>{ssn.text}</ssn>
        </registration>
    </mx:Model>

    <!-- Form contains user input controls. -->
    <mx:Form>
        <mx:FormItem label="Name" required="true">
            <mx:TextInput id="fullname" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Email" required="true">
            <mx:TextInput id="email" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Phone" required="true">
            <mx:TextInput id="phone" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Zip" required="true">
            <mx:TextInput id="zip" width="60"/>
        </mx:FormItem>

        <mx:FormItem label="Social Security" required="true">
            <mx:TextInput id="ssn" width="200"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Using ActionScript expressions in curly braces

Binding expressions in curly braces can contain an ActionScript expression that returns a value. For example, you can use the curly braces syntax for the following types of binding:

- A single bindable property inside curly braces

- String concatenation that includes a bindable property inside curly braces
- Calculations on a bindable property inside curly braces
- Conditional operations that evaluate a bindable property value

The following example shows a data model that uses each type of binding expression:

```
<?xml version="1.0"?>
<!-- binding/AsInBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Model id="myModel">
        <myModel>
          <!-- Perform simple property binding. -->
          <a>{nameInput.text}</a>
          <!-- Perform string concatenation. -->
          <b>This is {nameInput.text}</b>
          <!-- Perform a calculation. -->
          <c>{(Number(numberInput.text) as Number) * 6 / 7}</c>
         <!-- Perform a conditional operation using a ternary operator. -->
          <d>{(isMale.selected) ? "Mr." : "Ms."} {nameInput.text}</d>
        </myModel>
    </mx:Model>

    <mx:Form>
        <mx:FormItem label="Last Name:">
            <mx:TextInput id="nameInput"/>
        </mx:FormItem>
        <mx:FormItem label="Select sex:">
            <mx:RadioButton id="isMale"
                label="Male"
                groupName="gender"
                selected="true"/>
            <mx:RadioButton id="isFemale"
                label="Female"
                groupName="gender"/>
        </mx:FormItem>
        <mx:FormItem label="Enter a number:">
            <mx:TextInput id="numberInput" text="0"/>
        </mx:FormItem>
    </mx:Form>

    <mx:Text
     text="{'Calculation: '+numberInput.text+' * 6 / 7 = '+myModel.c}"/>
    <mx:Text text="{'Conditional: '+myModel.d}"/>
</mx:Application>
```

Using an E4X expression in curly braces

A binding expression in curly braces or an `<mx:Binding>` tag can contain an ECMAScript for XML (E4X) expression when the source of a binding is a bindable property of type XML. A binding expression in curly braces automatically calls the `toString()` method when the binding destination is a String property. A binding expression in curly braces or an `<mx:Binding>` tag can contain an ECMAScript for XML (E4X) expression when the source of a binding is a bindable property of type XML; for more information, see "Using an E4X expression in an <mx:Binding> tag" on page 1256.

In the code in the following example, there are three binding expressions in curly braces that bind data from an XML object. The first uses . (dot) notation, the second uses .. (dot dot) notation, and the third uses || (or) notation.

```
<?xml version="1.0"?>
<!-- binding/E4XInBraces.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            [Bindable]
            public var xdata:XML = <order>
                <item id = "3456">
                    <description>Big Screen Television</description>
                    <price>1299.99</price><quantity>1</quantity>
                </item>
                <item id = "56789">
                    <description>DVD Player</description>
                    <price>399.99</price>
                    <quantity>1</quantity>
                </item>
            </order>;
        ]]>
    </mx:Script>

    <mx:Label text="Using .. notation."/>
    <!-- Inline databinding will automatically call the
        toString() method when the binding destination is a string. -->
    <mx:List width="25%"
        dataProvider="{xdata..description}"/>

    <mx:Label text="Using . notation."/>
    <mx:List width="25%"
        dataProvider="{xdata.item.description}"/>

    <mx:Label text="Using || (or) notation."/>
    <mx:List width="25%"
```

```
dataProvider="{xdata.item.(@id=='3456'||@id=='56789').description}"/>
</mx:Application>
```

Using ActionScript functions in curly braces

You can use ActionScript functions as the source of binding expressions. You usually do this when using a bindable property as an argument of a function. When the bindable property changes, the function executes, and the result is used in the binding destination, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/ASInBraces.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            [Bindable]
            public var inString:String;

            public function getNewText(sourceString:String):String {
                return sourceString.toUpperCase();
            }
        ]]>
    </mx:Script>

    <mx:TextInput id="myTI" text="Enter text here"/>
    <mx:Button label="Trigger Binding" click="inString=myTI.text;"/>

    <mx:TextArea text="{getNewText(inString)}"/>
</mx:Application>
```

In this example, Flex calls the `getNewText()` function to update the TextArea control every time the `inString` property is updated.

If the `inString` property is not passed as an argument, but is referenced from within the function, the function does not get invoked when the `inString` property changes. In the following example, the `getNewText()` references the `inString` property:

```
public function getNewText():String {
  if(inString == "")
  ...
}
```

You then use the function in a binding expression:

```
<mx:TextArea text="{getNewText()}"/>
```

The `getNewText()` function gets called once when the applications starts, but changes to the `inString` property do not trigger a data binding, and the TextArea controls remains unchanged.

Binding data with the <mx:Binding> tag

You can use the `<mx:Binding>` tag as an alternative to the curly braces syntax. When you use the `<mx:Binding>` tag, you provide a source property in the `<mx:Binding>` tag's `source` property and a destination property in its `destination` property. This is equivalent to using the curly braces syntax.

In contrast with the curly braces syntax, you can use the `<mx:Binding>` tag to completely separate the view (user interface) from the model. The `<mx:Binding>` tag also lets you bind different source properties to the same destination property because you can specify multiple `<mx:Binding>` tags with the same destination.

In the following example, the properties of user interface controls are bound to the myEmployee data model using <mx:Binding> tags:

```
<?xml version="1.0"?>
<!-- binding/BindingTags.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- The myEmployee data model. -->
    <mx:Model id="myEmployee">
        <Employee>
            <name>
                <first/>
                <last/>
            </name>
            <department/>
            <email/>
        </Employee>
    </mx:Model>

    <!-- Properties of user interface controls are bound to the
        myEmployee data model using <mx:Binding> tags. -->
    <mx:Binding source="firstName.text"
        destination="myEmployee.name.first"/>
    <mx:Binding source="lastName.text"
        destination="myEmployee.name.last"/>
    <mx:Binding source="department.text"
        destination="myEmployee.department"/>
    <mx:Binding source="email.text"
        destination="myEmployee.email"/>

    <!-- Form contains user input controls. -->
    <mx:Form label="Employee Information">
        <mx:FormItem label="First Name">
            <mx:TextInput id="firstName"/>
        </mx:FormItem>
        <mx:FormItem label="Last Name">
            <mx:TextInput id="lastName"/>
        </mx:FormItem>
        <mx:FormItem label="Department">
            <mx:TextInput id="department"/>
        </mx:FormItem>
        <mx:FormItem label="Email Address">
            <mx:TextInput id="email"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Using ActionScript expressions in Binding tags

The `source` property of an `<mx:Binding>` tag can contain curly braces. When there are no curly braces in the `source` property, the value is treated as a single ActionScript expression. When there are curly braces in the `source` property, the value is treated as a concatenated ActionScript expression. The `<mx:Binding>` tags in the following example are valid and equivalent to each other:

```
<?xml version="1.0"?>
<!-- binding/ASInBindingTags.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function whatDogAte():String {
                return "homework";
            }
        ]]>
    </mx:Script>

    <mx:Binding
        source="'The dog ate my '+ whatDogAte()"
        destination="field1.text"/>
    <mx:Binding
        source="{'The dog ate my '+ whatDogAte()}"
        destination="field2.text"/>
    <mx:Binding
        source="The dog ate my {whatDogAte()}"
        destination="field3.text"/>

    <mx:TextArea id="field1"/>
    <mx:TextArea id="field2"/>
    <mx:TextArea id="field3"/>
</mx:Application>
```

The `source` property in the following example is not valid because it is not an ActionScript expression:

```
<mx:Binding source="The dog ate my homework" destination="field1.text"/>
```

Binding more than one source property to a destination property

You can bind more than one source property to the same destination property by using multiple `<mx:Binding>` tags that specify the same destination but different sources, or by using the combination of binding expressions in curly braces and `<mx:Binding>` tags. You cannot do this with just the curly braces syntax.

In the following example, the data model field `thing1.part` is the destination, and both `input1.text` and `input2.text` are its sources. If `input1.text` or `input2.text` is updated, `thing1.part` contains the updated value.

```
<?xml version="1.0"?>
<!-- binding/BindMultSources.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Model id="thing1">
        <data>
            <part>{input1.text}</part>
        </data>
    </mx:Model>

    <mx:Binding source="input2.text" destination="thing1.part"/>

    <mx:TextInput id="input1"/>
    <mx:TextInput id="input2"/>

    <mx:TextArea text="{thing1.part}"/>
</mx:Application>
```

Binding a source property to more than one destination property

You can bind a single source property to more than one destination property. In the following example, a TextInput control's `text` property is bound to properties of two data models, and the data model properties are bound to the `text` properties of two Label controls.

```
<?xml version="1.0"?>
<!-- binding/BindMultDestinations.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Model id="mod1">
        <data>
            <part>{input1.text}</part>
        </data>
    </mx:Model>

    <mx:Model id="mod2">
        <data>
            <part>{input1.text}</part>
        </data>
    </mx:Model>

    <mx:TextInput id="input1" text="Hello" />

    <mx:Label text="{mod1.part}"/>
    <mx:Label text="{mod2.part}"/>
</mx:Application>
```

Using an E4X expression in an ‹mx:Binding› tag

A binding expression curly braces or an `<mx:Binding>` tag can contain an ECMAScript for XML (E4X) expression when the source of a binding is a bindable property of type XML. Unlike an E4X expression in curly braces, when you use an E4X expression in an `<mx:Binding>` tag, you must explicitly call the `toString()` method when the binding destination is a String property.

In the code in the following example, there are three binding expressions in curly braces that bind data from an XML object. The first uses . (dot) notation, the second uses .. (dot dot) notation, and the third uses || (or) notation.

```
<?xml version="1.0"?>
<!-- binding/E4XInBindingTag.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600" height="900">

    <mx:Script>
        <![CDATA[
            [Bindable]
            public var xdata:XML =
                <order>
                    <item id = "3456">
                        <description>Big Screen Television</description>
                        <price>1299.99</price><quantity>1</quantity>
                    </item>
                    <item id = "56789">
                        <description>DVD Player</description>
                        <price>399.99</price>
                        <quantity>1</quantity>
                    </item>
                </order>;
        ]]>
    </mx:Script>

    <mx:Label text="Using .. notation."/>

    <!-- This will update because what is
        binded is actually the String and XMLList. -->
    <mx:List width="75%" id="txts"/>
    <mx:Binding
        source="xdata..description"
        destination="txts.dataProvider"/>

    <mx:Label text="Using . notation."/>
    <mx:List width="75%" id="txt2s"/>
    <mx:Binding
        source="xdata.item.description"
        destination="txt2s.dataProvider"/>

    <mx:Label text="Using || (or) notation."/>
    <mx:List width="75%" id="txt3s"/>
    <mx:Binding
        source="xdata.item.(@id=='3456'||@id=='56789').description"
        destination="txt3s.dataProvider"/>
</mx:Application>
```

About the binding mechanism

At compile time, the MXML compiler generates code to create ActionScript Watcher and Binding objects that correspond to the binding tags and expressions found in an MXML document. At run time, Watcher objects are triggered by change events that come from the constituent parts of binding source expressions; the Watcher objects then trigger Binding objects to execute bindings.

To ensure the best possible binding results, you should always strongly type your variables. When you use one of the standard Flex classes, such as any of the list-based classes, you should know that the `selectedItem` property is typed as Object. If your `selectedItem` is a real custom class, not a model or RPC service result, you should cast it, as the binding expression in the following example shows:

```
{MyClass(myList.selectedItem).someProp}
```

Working with bindable property chains

When you specify a property as the source of a data binding, Flex monitors not only that property for changes, but also the chain of properties leading up to it. The entire chain of properties, including the destination property, is called a *bindable property chain*. In the following example, `firstName.text` is a bindable property chain that includes both a `firstName` object and its `text` property:

```
<first>{firstName.text}</first>
```

You should raise an event when any named property in a bindable property chain changes. If the property is marked with the `[Bindable]` metadata tag, the Flex compiler generates the event for you. For more examples using the `[Bindable]` metadata tag, see "Bindable metadata tag" in *Creating and Extending Flex 2 Components*.

The following example shows an example that uses the `[Bindable]` metadata tag for a variable and a getter property and how to call the `dispatchEvent()` function:

```
[Bindable]
public var minFontSize:Number = 5;


[Bindable("textChanged")]
public function get text():String {
   return myText;
}

public function set text(t : String):void {
   myText = t;
  dispatchEvent( new Event( "textChanged" ) );}
```

If you omit the event name in the `[Bindable]` metadata tag, the Flex compiler automatically generates and dispatches an event named `propertyChange` so that the property can be used as the source of a data binding expression.

You should also provide the compiler with better information about an object by casting the object to a known type. In the following example, the `myList` List control contains Customer objects, so the `selectedItem` property is cast to a Customer object:

```
<mx:Model id="selectedCustomer">
  <customer>
    <name>{Customer(myList.selectedItem).name}</name>
    <address>{Customer(myList.selectedItem).address}</address>
    ...
  </customer>
</mx:Model>
```

There are some situations in which binding does not execute automatically as expected. Binding does not execute automatically when you change an entire item of a `dataProvider` property.

Binding also does not execute automatically for subproperties of properties that have `[Bindable]` metadata, as the following example shows:

```
...
[Bindable]
var temp;
// Binding is triggered:
temp = new Object();
// Binding is not triggered, because label not a bindable property
// of Object:
temp.label = foo;
...
```

In this code example, the problem with `{temp.label}` is that temp is an Object. You can solve in one of the following ways:

- Preinitialize the Object.
- Assign an ObjectProxy to `temp`; all of an ObjectProxy's properties are bindable.
- Make `temp` a strongly typed object with a `label` property that is bindable.

Binding also does not execute automatically when you are binding data to a property that Flash Player updates automatically, such as the `mouseX` property.

The `executeBindings()` method of the UIComponent class executes all the bindings for which a UIComponent object is the destination. All containers and controls, as well as the Repeater component, extend the UIComponent class. The `executeChildBindings()` method of the Container and Repeater classes executes all of the bindings for which the child UIComponent components of a Container or Repeater class are destinations. All containers extend the Container class.

These methods give you a way to execute bindings that do not occur as expected. By adding one line of code, such as a call to `executeChildBindings()` method, you can update the user interface after making a change that does not cause bindings to execute. However, you should only use the `executeBindings()` method when you are sure that bindings do not execute automatically.

Defining bindings in ActionScript

Help ID:

You typically define a data binding in MXML by using the curly braces ({ }) or the `<mx:Binding>` tag. You can also define a binding in ActionScript by using the mx.binding.utils.BindingUtils class. This class defines static methods that let you create a binding to a property implemented as a variable, by using the `bindProperty()` method, or to a property implemented as a setter method, by using the `bindSetter()` method.

In the following example, you use the `bindProperty()` method to create a binding between a TextInput control and a TextArea control:

```
<?xml version="1.0"?>
<!-- binding/BindInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initBinding();">

    <mx:Script>
      <![CDATA[

        import mx.binding.utils.*;

        // Define data binding.
        public function initBinding():void {
      BindingUtils.bindProperty(textarea, "text", textinput, "text");
        }
      ]]>
    </mx:Script>

    <mx:TextInput id="textinput" text="Hello"/>
    <mx:TextArea id="textarea"/>
    <mx:Button label="Submit" click="textinput.text='Goodbye';"/>
</mx:Application>
```

The following example uses the `bindSetter()` method to set up two bindings. As you enter text in the TextInput controls, it is copied to the corresponding TextArea control:

```
<?xml version="1.0"?>
<!-- binding/BindSetterAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.binding.utils.*;
            import mx.events.FlexEvent;

            // Set method.
            public function setMyString(val:String):void {
                taSetter2.text = val;
            }

            <!-- Event listener to configure binding with a setter. -->
            public function mySetterBindingInline(event:FlexEvent):void {
                var watcherSetter:ChangeWatcher =
                    BindingUtils.bindSetter(
                        function(v:String):void {
                            taSetter1.text = v}, tiSetter1, "text");
            }

            <!-- Event listener to configure binding with a setter. -->
            public function mySetterBinding(event:FlexEvent):void {
                var watcherSetter:ChangeWatcher =
                   BindingUtils.bindSetter(setMyString, tiSetter2, "text");
            }
        ]]>
    </mx:Script>

    <mx:Label text="Bind Setter using inline setter"/>
    <mx:TextInput id="tiSetter1" text="Hello Setter" />
    <mx:TextArea id="taSetter1"
        initialize="mySetterBindingInline(event);"/>

    <mx:Label text="Bind Setter using setter method"/>
    <mx:TextInput id="tiSetter2" text="Hello Setter" />
    <mx:TextArea id="taSetter2"
        initialize="mySetterBinding(event);"/>
</mx:Application>
```

Defining binding watchers

You can detect when a binding occurs in your application. Flex includes the mx.binding.utils.ChangeWatcher class that you can use to define a watcher for a data binding. Typically, your event watcher invokes an event listener when the binding occurs.

To set up a binding watcher, you use the static `watch()` method of the ChangeWatcher class, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/DetectWatcher.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initWatcher();">

    <mx:Script>
      <![CDATA[
        import mx.utils.*;
        import mx.binding.utils.*;
        import mx.events.FlexEvent;
        import mx.events.PropertyChangeEvent;

        // Define a bindable property.
        [Bindable]
        public var propChain:Object;

        public var myWatcher:ChangeWatcher;

        // Define binding watcher.
        public function initWatcher():void {
            // Define a watcher for the text binding.
            ChangeWatcher.watch(textarea, "text", watcherListener);

            // Initialize the Object.
            propChain = new ObjectProxy();
            propChain.sub1 = new ObjectProxy();
            propChain.sub1.sub2 = new String("first sub2");

            // Define a watcher for the Object binding.
            myWatcher=
                ChangeWatcher.watch(this, ["propChain","sub1","sub2"],
                    watcherListenerProp);
        }

        // Event listener when binding occurs.
        public function watcherListener(event:Event):void {
            myTA1.text="binding occurred";
        }

        // Event listener when binding occurs.
        public function watcherListenerProp(event:PropertyChangeEvent):void
```

```
        {
            myTA2.text="prop chain binding occurred";
            myWatcher.unwatch();
        }
      ]]>
    </mx:Script>

    <!-- Define a binding expression to watch. -->
    <mx:TextInput id="textinput" text="Hello"/>
    <mx:TextArea id="textarea" text="{textinput.text}"/>

    <!-- Trigger a binding. -->
    <mx:Button label="Submit" click="textinput.text='Goodbye';"/>
    <mx:TextArea id="myTA1"/>

    <!-- Trigger a binding to a bindable property chain. -->
    <mx:Button label="Submit prop chain"
        click="propChain.sub1.sub2 = 'second sub2';"/>
    <mx:TextArea id="myTA2"/>
</mx:Application>
```

This example uses the ObjectProxy class to manipulate the propChain variable because that class provides the ability to track changes to an item. In this example, the event listener for the propChain Object uses the `unwatch()` method to remove the watcher after the first binding occurs.

You define an event listener for each binding watcher, where the event listeners take a single argument that contains the event object. The data type of the event object is determined by the property being watched. Each bindable property can dispatch a different event type and the associated event object. The `[Bindable]` metadata tag, used to make a property bindable, specifies the event type of the event dispatched when the property changes. For more information about the `[Bindable]` metadata tag, see Chapter 5, "Using Metadata Tags in Custom Components," in *Creating and Extending Flex 2 Components*.

The watcherListener() event listener takes an event object of type FlexEvent because that is the data type of the event dispatched by the `text` property of the TextArea control when it changes.

The watcherListenerProp() event listener takes an event object of type PropertyChangeEvent because that is the data type of the event dispatched by the propChain Object. This is also the event type associated with the `propertyChange` event, the default event dispatched by a property when you do not specify an event type in the `[Bindable]` metadata tag.

For both event listeners, you can specify an event type of flash.events.Event, the base class for all Flex events.

Considerations for using the binding feature

Consider the following when using the binding feature:

- Because the SharedObject object is built into Adobe Flash Player, its behavior cannot be modified by the binding mechanism. To use binding with a SharedObject, you must write a wrapper object that uses a SharedObject object internally, but has getter/setter properties that the rest of your application can use. If you do not use a wrapper object, a SharedObject appears to lose data, specifically data that might have been used in a binding expression.
- You must use casting when binding to an object in which a variable is typed as Object if the object has a getter/setter property. This affects Repeater objects and other objects with a `dataProvider` property. For more information about casting, see "Working with bindable property chains" on page 1258.
- The `selectedItem` property of the List class and the `currentItem` property of Repeater class are typed explicitly as Object. Binding works correctly when the property is a simple object with name-value pairs. However, binding fails if the property is a typed object in which a bound property is a getter/setter property. Because the variable is explicitly typed to Object, no warnings are issued. When binding to the `selectedItem` or `currentItem` property, and the property that you are binding to is implemented as a getter/setter, cast to the appropriate class, as the following example shows:

  ```
  {MyGetterClass(theList.selectedItem).myGetterProp}
  ```
- If a component uses binding expressions, the bindings execute before the component's change event executes, and destinations are updated before their component's change event executes. However, you could bind a component to an object that you haven't created yet. If you must rely on all bindings having run before your code fragment executes, you can use the root tag's event.
- You cannot bind to styles.
- If you bind a model into the `dataProvider` property of a component, you should not change items in the model directly. Instead, change the items through the Collections API. Otherwise, the component to which the model is bound is not redrawn to show the changes to the model. For example, instead of using the following:

  ```
  myGrid.getItemAt(itemIndex).myField = 1;
  ```

  You would use the following:

  ```
  myGrid.dataProvider.editField(itemIndex, "myField", 1);
  ```

- Array elements cannot function as binding sources at run time. Arrays that are bound do not stay updated if individual fields of a source Array change. Binding copies values during instantiation after variables are declared in an `<mx:Script>` tag, but before event listeners execute.

Debugging data binding

In some situations, data binding may not appear to function correctly, and you may need to debug them. The following list contains suggestions for resolving data binding issues:

- Pay attention to warnings.

  It is easy to see a warning and think that it doesn't matter, especially if binding appears to work at startup, but warnings are important.

  If a warning is about a missing `[Bindable]` on a getter/setter property, even if the binding works at startup, subsequent changes to the property are not noticed.

  If a warning is about a static variable or a built-in property, changes aren't noticed.

  If you are sure that the property should be bindable but the compiler is generating warnings about an unknown type, you might need to cast an object in your binding expression so that the compiler can find the appropriate type information, including `[Bindable]` metadata. For more information about casting, see "Working with bindable property chains" on page 1258.
- Ensure that the source of the binding actually changed.

  When your source is part of a larger procedure, it is easy to miss the fact that you never assigned the source.
- Ensure that the bindable event is being dispatched.

  You can use the Flex command-line debugger (fdb), or the Adobe Flex Builder debugger to make sure that the `dispatchEvent()` method is called. Also, you can add a normal event listener to that class to make sure it gets called. If you want to add the event listener as a tag attribute, you must place the `[Event('myEvent')]` metadata at the top of your class definition or in an `<mx:Metadata>` tag in your MXML.
- Create a setter function and use a `<mx:Binding>` tag to assign into it.

  You can then put a trace or an alert or some other debugging code in the setter with the value that is being assigned. This technique ensures that the binding itself is working. If the setter is called with the right information, you will know that it's your destination that is failing, and you can start debugging there.

Using binding for moving related data

Many applications require a messaging strategy for moving related data between objects. You can use binding with messaging objects, whose sole purpose is to move related data around an application.

For example, suppose you are building a library card catalog system, and you want to let users search a library database. The first thing you do is to provide the users with a search form. At the same time, you create a simple Query object in an ActionScript class. The Query object has fields for author, title, and subject. Because the Query object holds only typed data, it does not need a lot of additional functionality.

The user is going to create only one query at a time, so you declare the Query object as a custom ActionScript component inside an MXML component. You then bind the search form elements into the properties of the Query object, as the following example shows:

```
<?xml version="1.0"?>
<!-- QueryForm.mxml -->

<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:library="*">
  <library:Query id="myQuery">
    <library:author>{authorInput.text}</library:author>
    <library:title>{titleInput.text}</library:title>
    <library:subject>{subjectInput.text}</library:subject>
  </library:Query>

  <mx:Form>
    <mx:FormItem label="Author">
      <mx:TextInput id="authorInput"/>
    </mx:FormItem>
  ...
  </mx:Form>
    <mx:Button label="Submit Query"
  click="queryBizDelegate.sendQuery(myQuery)"/>
</mx:VBox>
```

You could also include validators and any other user interface logic that you require in this form. The main idea is that instead of having to know the parameters of the `sendQuery()` method, you can pass a Query object, which is filled in by using binding, to the form.

Going in the other direction, suppose the library query service returns a result that contains a collection of books. Some separate logic can navigate to the results page, and that page might have a DataGrid control to display all the books that are returned. The service can simply drop off the books at a known location. It does not need to know that there is a view that cares about the result. The following example shows the service result listener:

```
public function queryResult(event):void {
  queryBizDelegate.acceptResult(event.result);
}
```

The `queryBizDelegate.acceptResult()` method stores the result in a well-known location, such as a `queryResults` property. You can bind the `queryResults` property to a DataGrid control's `dataProvider` property so that the view automatically updates whenever new results arrive, as the following example shows:

```
<mx:DataGrid dataProvider="{queryBizDelegate.queryResults}"/>
```

CHAPTER 39

Storing Data

39

This topic describes the data model feature, which you use to store data in an application.

Contents

About data models

A *data model* is an ActionScript object that contains properties that you use to store application-specific data. Communication between an Adobe Flex application and the server is required only to retrieve data not yet available to the Flex application and to update a server-side data source with new data from the Flex application. Data models provide a way to store data in the Flex application before it is sent to the server, or to store data sent from the server before using it in the application.

You can use a data model for data validation, and it can contain client-side business logic. You can define a data model in MXML or ActionScript. In the model-view-controller (MVC) design pattern, the data model represents the model tier.

When you plan an application, you determine the kinds of data that the application must store and how that data must be manipulated. This helps you decide what types of data models you need. For example, suppose you decide that your application must store data about employees. A simple employee model might contain name, department, and e-mail address properties.

Defining a data model

You can define a data model in an MXML tag, an ActionScript function, or an ActionScript class. In general, you should use MXML-based models for simple data structures, and use ActionScript for more complex structures and client-side business logic.

NOTE

The `<mx:Model>` and `<mx:XML>` tags are Flex compiler tags and do not correspond directly to ActionScript classes. The *Adobe Flex 2 Language Reference* contains information about these tags and other compiler tags. Click the Appendixes link on the main page in the *Adobe Flex 2 Language Reference*.

You can place an `<mx:Model>` tag or an `<mx:XML>` tag in a Flex application file or in an MXML component file. The tag should have an `id` value, and it cannot be the root tag of an MXML component.

The <mx:Model> tag

The most common type of MXML-based model is the `<mx:Model>` tag, which is compiled into an ActionScript object of type mx.utils.ObjectProxy, which contains a tree of objects when your data is in a hierarchy, with no type information. The leaves of the Object tree are scalar values. Because models that are defined in `<mx:Model>` tags contain no type information or business logic, you should use them only for the simplest cases. Define models in ActionScript classes when you need the typed properties or you want to add business logic.

The following example shows an employee model declared in an `<mx:Model>` tag:

```
<?xml version="1.0"?>
<!-- Models\ModelTag.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Model id="employeemodel">
        <employee>
            <name>
                <first/>
                <last/>
            </name>
            <department/>
            <email/>
        </employee>
    </mx:Model>
</mx:Application>
```

An `<mx:Model>` child tag with no value is considered null. If you want an empty string instead, you can use a binding expression as the value of the tag, as the following example shows:

<?xml version="1.0"?>

```
<!-- Models\ModelTagEmptyString.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
  <mx:Model id="employeemodel">
    <employee>
      <name>
        <!--Fill the first property with empty string.-->
        <first>{""}</first>
        <!--Fill the last property with empty string.-->
        <last>{""}</last>
      </name>
      <!--department is null-->
      <department/>
      <!--email is null-->
      <email/>
    </employee>
  </mx:Model>
</mx:Application>
```

You could use an ActionScript class to work around this limitation of the `<mx:Model>` tag.

The <mx:XML> tag

An `<mx:XML>` tag represents literal XML data. Setting the `format` property to `e4x` creates an XML object, which implements the powerful XML-handling standards defined in the ECMAScript for XML specification (ECMA-357 edition 2) (known as E4X). For backward compatibility, when the `format` property is not explicitly set to `e4x`, the type of the object created is flash.xml.XMLNode.

NOTE	You currently cannot use a node within an `<mx:XML>` data model as a binding source.

Script-based models

As an alternative to using an MXML-based model, you can define a model as a variable in an `<mx:Script>` tag. The following example shows a very simple model defined in an ActionScript script block. It would be easier to declare this model in an `<mx:Model>` tag.

```
<?xml version="1.0"?>
<!-- Models\ScriptModel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Script>
        <![CDATA[
            [Bindable]
            public var myEmployee:Object={
                name:{
                    first:null,
                    last:null
                    },
                department:null,
                email:null
            };
        ]]>
    </mx:Script>
</mx:Application>
```

There is no advantage of using a script-based model instead of an MXML-based model. As with MXML-based models, you cannot type the properties of a script-based model. To type properties, you must use a class-based model.

Class-based models

Using an ActionScript class as a model is a good option when you want to store complex data structures with typed properties, or when you want to execute client-side business logic using application data. Also, the type information in a class-based model is retained on the server when the model is passed to a server-side data service.

The following example shows a model defined in an ActionScript class. This model is used to store shopping cart items in an e-commerce application. It also provides business logic in the form of methods for adding and removing items, getting an item count, and getting the total price. For more information on ActionScript components, see *Creating and Extending Flex 2 Components*.

```
package
{

[Bindable]
public class ShoppingCart {
```

```
    public var items:Array = [];

    public var total:Number = 0;

    public var shippingCost:Number = 0;

    public function ShoppingCart() {
    }

    public function addItem(item:Object, qty:int = 1,
      index:int = 0):void {
      items.splice(index, 0, { id: item.id,
      name: item.name,
      description: item.description,
      image: item.image,
      price: item.price,
      qty: qty });
      total += parseFloat(item.price) * qty;
     }

    public function removeItemAt(index:Number):void {
      total -= parseFloat(items[index].price) * items[index].qty;
      items.splice(index, 1);
      if (getItemCount() == 0)
      shippingCost = 0;
     }

    public function getItemCount():int {
      return items.length;
    }

    public function getTotal():Number {
      return total;
    }
}
}
```

NOTE You can use properties of any recognized type in a class-based model. For example, you could create an Employee class, and then define properties of type Employee in another class that imports Employee.

You declare a class-based model as an ActionScript component tag in an MXML file, as the following example shows:

```
<local:ShoppingCart id="cart" xmlns:local="*"/>
```

This component is in the same directory as the MXML file, as indicated by the XML namespace value `*`. For more information about specifying the location of components, see *Creating and Extending Flex 2 Components*.

Specifying an external source for an <mx:Model> tag or <mx:XML> tag

You can specify an external source for an `<mx:Model>` or `<mx:XML>` tag in a `source` property. Separating the content of a model from the MXML that defines the user interface improves the maintainability and reusability of an application. Adobe recommends this way of adding static XML content to a Flex application.

The external source file can contain static data and data binding expressions, just like a model defined in the body of the `<mx:Model>` or `<mx:XML>` tag. The file referenced in a `source` property resides on the server and not on the client machine. The compiler reads the `source` value and compiles the source into the application; the `source` value is not read at run time. To retrieve XML data at run time, you can use the `<mx:HTTPService>` tag; for more information, see Chapter 45, "Using RPC Components," on page 1407.

Using `<mx:Model>` and `<mx:XML>` tags with external sources is an easy way to reuse data model structures and data binding expressions. You can also use them to prepopulate user interface controls with static data by binding data from the model elements into the user interface controls.

The `source` property accepts the names of files relative to the current web application directory, as well as URLs with HTTP:// prefixes. In the following example, the content of the myEmployee1 data model is an XML file named content.xml in the local web application directory. The content of the myEmployee2 data model is a fictional HTTP URL that returns XML.

```
<mx:Model source="employees.xml" id="employee1"/>

<mx:Model source="http://www.somesitel.com/employees.xml" id="employee2"/>
```

The source file must be a valid XML document with a single root node. The following example shows an XML file that could be used as the source of the `<mx:Model source="employees.xml" id="Model1"/>` tag. <

```
<?xml version="1.0"?>
<employees>
  <employee>
       <name>John Doe</name>
       <phone>555-777-66555</phone>
       <email>jdoe@fictitious.com</email>
       <active>true</active>
    </employee>
    <employee>
       <name>Jane Doe</name>
       <phone>555-777-66555</phone>
       <email>jndoe@fictitious.com</email>
       <active>true</active>
    </employee>
</employees>
```

Using validators with a data model

To validate the data stored in a data model, you use validators. In the following example, the `<mx:EmailValidator>`, `<mx:PhoneNumberValidator>`, `<mx:ZipCodeValidator>`, and `<mx:SocialSecurityValidator>` tags declare validators that validate the email, phone, zip, and ssn fields of the registration data model. The validators generate error messages when a user enters incorrect data in TextInput controls that are bound to the data model fields.

```
<?xml version="1.0"?>
<!-- Models\ModelTagEmptyString.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Model id="reg">
        <registration>
            <name>{username.text}</name>
            <email>{email.text}</email>
            <phone>{phone.text}</phone>
            <zip>{zip.text}</zip>
            <ssn>{ssn.text}</ssn>
        </registration>
    </mx:Model>

    <mx:Validator required="true" source="{reg}" property="name"
        trigger="{submit}" triggerEvent="click" listener="{username}"/>
    <mx:EmailValidator source="{reg}" property="email"
        trigger="{submit}" triggerEvent="click" listener="{email}"/>
    <mx:PhoneNumberValidator source="{reg}" property="phone"
        trigger="{submit}" triggerEvent="click" listener="{phone}"/>
    <mx:ZipCodeValidator source="{reg}"
        property="zip" trigger="{submit}" triggerEvent="click"
listener="{zip}"/>
    <mx:SocialSecurityValidator source="{reg}" property="ssn"
        trigger="{submit}" triggerEvent="click" listener="{ssn}"/>


    <!-- Form contains user input controls. -->
    <mx:Form>
        <mx:FormItem label="Name" required="true">
            <mx:TextInput id="username" width="200"/>
        </mx:FormItem>
        <mx:FormItem label="Email" required="true">
            <mx:TextInput id="email" width="200"/>
        </mx:FormItem>
        <mx:FormItem label="Phone" required="true">
            <mx:TextInput id="phone" width="200"/>
        </mx:FormItem>
        <mx:FormItem label="Zip" required="true">
            <mx:TextInput id="zip" width="60"/>
        </mx:FormItem>
```

```
        <mx:FormItem label="Social Security" required="true">
            <mx:TextInput id="ssn" width="200"/>
        </mx:FormItem>
        <mx:FormItem>
        <!-- User clicks Button to trigger validation. -->
            <mx:Button id="submit" label="Validate"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

This example cleanly separates the user interface and application-specific data. You could easily extend it to create a three-tier architecture by binding data from the registration data model into an RPC service request. You could also bind user input data directly into an RPC service request, which itself is a data model, as described in Chapter 45, "Using RPC Components," on page 1407.

For more information about validators, see Chapter 40, "Validating Data," on page 1281.

Using a data model as a value object

You can use a data model as a value object, which acts as a central repository for a set of data returned from method calls on one or more objects. This makes it easier to manage and work with data in an application.

In the following example, the tentModel data model stores the results of a web service operation. The TentDetail component is a custom MXML component that gets its data from the tentModel data model and displays details for the currently selected tent.

```
...
<!-- Data model stores data from selected tent. -->
<mx:Model id="tentModel">
  <tent>
    <name>{selectedTent.name}</name>
    <sku>{selectedTent.sku}</sku>
    <capacity>{selectedTent.capacity}</capacity>
    <season>{selectedTent.seasonStr}</season>
    <type>{selectedTent.typeStr}</type>
    <floorarea>{selectedTent.floorArea}</floorarea>
    <waterproof>{getWaterProof(selectedTent.waterProof)}</waterproof>
    <weight>{getWeight(selectedTent)}</weight>
    <price>{selectedTent.price}</price>
  </tent>
</mx:Model>
...
  <TentDetail id="detail" tent="{tentModel}"/>
...
```

The following example shows the MXML source code for the TentDetail component. References to the `tent` property, which contains the tentModel data model, and the corresponding `tentModel` properties are highlighted in boldface font.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml" title="Tent Details">

  <mx:Script>
    <![CDATA[
      [Bindable]
      public var tent:Object;
    ]]>
  </mx:Script>

  <mx:Style>
    .title{fontFamily:Arial;fontWeight:bold;color:#3D3D3D;fontSize:16pt;}
    .flabelColor
      {fontFamily:Arial;fontWeight:bold;color:#3D3D3D;fontSize:11pt}
    .productSpec{fontFamily:Arial;color:#5B5B5B;fontSize:10pt}
  </mx:Style>

  <mx:VBox paddingLeft="10" paddingTop="10" paddingRight="10">

    <mx:Form verticalGap="0" paddingLeft="10" paddingTop="10"
      paddingRight="10" paddingBottom="0">

      <mx:VBox width="209" height="213">
        <mx:Image width="207" height="211"
          source="./images/{tent.sku}_detail.jpg"/>
      </mx:VBox>

      <mx:FormHeading label="{tent.name}" paddingTop="1"
        styleName="title"/>

      <mx:HRule width="209"/>

      <mx:FormItem label="Capacity" styleName="flabelColor">
        <mx:Label text="{tent.capacity} person"
          styleName="productSpec"/>
      </mx:FormItem>
      <mx:FormItem label="Season"
        styleName="flabelColor">
        <mx:Label text="{tent.season}"
          styleName="productSpec"/>
      </mx:FormItem>
      <mx:FormItem label="Type" styleName="flabelColor">
        <mx:Label text="{tent.type}"
          styleName="productSpec"/>
      </mx:FormItem>
      <mx:FormItem label="Floor Area" styleName="flabelColor">
```

```
        <mx:Label text="{tent.floorarea}
          square feet" styleName="productSpec"/>
      </mx:FormItem>
      <mx:FormItem label="Weather" styleName="flabelColor">
        <mx:Label text="{tent.waterproof}"
          styleName="productSpec"/>
      </mx:FormItem>
      <mx:FormItem label="Weight" styleName="flabelColor">
        <mx:Label text="{tent.weight}"
          styleName="productSpec"/>
      </mx:FormItem>
    </mx:Form>
  </mx:VBox>
</mx:Panel>
```

Binding data into an XML data model

Flex compiles the `<mx:XML>` tag into literal XML data in an ActionScript xml.XMLNode or XML object. This is different from the `<mx:Model>` tag, which Flex compiles into an Action object that contains a tree of ActionScript objects. To bind data into an `<mx:XML>` data model, you can use the curly braces syntax the same way you do with other data models. However, you cannot use a node within the data model as a binding source.

Adobe does not recommend using the `<mx:Binding>` tag for this type of binding because doing so requires you to write an appropriate ActionScript XML command as the `destination` property of the `<mx:Binding>` tag. For more information about the `<mx:XML>` tag, see "Defining a data model" on page 1270.

The following example shows an `<mx:XML>` data model with binding destinations in curly braces:

```
<?xml version="1.0"?>
<!-- Models\XMLBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:XML id="myEmployee" format="e4x">
    <employee>
      <name>
        <first>{firstName.text}</first>
        <last>{lastName.text}</last>
        </name>
      <department>{department.text}</department>
      <email>{email.text}</email>
```

```
        </employee>
    </mx:XML>
    <mx:TextInput id="firstName"/>
    <mx:TextInput id="lastName"/>
    <mx:TextInput id="department"/>
    <mx:TextInput id="email"/>
</mx:Application>
```

NOTE

You cannot bind one piece of XML into another.

CHAPTER 40

Validating Data

40

This topic describes the Adobe Flex 2 data validation mechanism, which lets you validate the data in an application. Flex provides predefined validators for many common types of user-supplied data, such as date, number, and currency values.

You often use Flex validators with data models. For more information about data models, see Chapter 39, "Storing Data," on page 1269.

Contents

Validating data

The data that a user enters in a user interface might or might not be appropriate to the application. In Flex, you use a *validator* to ensure the values in the fields of an object meet certain criteria. For example, you can use a validator to ensure that a user enters a valid phone number value, to ensure that a String value is longer than a set minimum length, or ensure that a ZIP code field contains the correct number of digits.

In typical client-server environments, data validation occurs on the server after data is submitted to it from the client. One advantage of using Flex validators is that they execute on the client, which lets you validate input data before transmitting it to the server. By using Flex validators, you eliminate the need to transmit data to and receive error messages back from the server, which improves the overall responsiveness of your application.

NOTE	Flex validators do not eliminate the need to perform data validation on the server, but provide a mechanism for improving performance by performing some data validation on the client.

Flex includes a set of validators for common types of user input data, such as ZIP codes, phone numbers, and credit card numbers. Flex includes the following validators:

- Using the CreditCardValidator class
- Using the CurrencyValidator class
- Using the DateValidator class
- Using the EmailValidator class
- Using the NumberValidator class
- Using the PhoneNumberValidator class
- Using the RegExpValidator class
- Using the SocialSecurityValidator class
- Using the StringValidator class
- Using the ZipCodeValidator class

About validators

You define validators using MXML or ActionScript. You declare a validator in MXML using the `<mx:Validator>` tag or the tag for the appropriate validator type. For example, to declare the standard PhoneNumberValidator validator, you use the `<mx:PhoneNumberValidator>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidator.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define the PhoneNumberValidator. -->
    <mx:PhoneNumberValidator id="pnV"
        source="{phoneInput}" property="text"/>

    <!-- Define the TextInput control for entering the phone number. -->
    <mx:TextInput id="phoneInput"/>
    <mx:TextInput id="zipCodeInput"/>
</mx:Application>
```

In the previous example, you enter a value into the TextInput control for the phone number. When you remove focus from the TextInput control by selecting the TextInput control for the ZIP code, the validator executes.

You use the `source` property of the validator to specify an object, and the `property` property to specify a field of the object to validate. For more information on the `source` and `property` properties, see "About the source and property properties" on page 1284.

Validator tags must always be immediate children of the root tag of an MXML file. In the previous example, the validator ensures that the user enters a valid phone number in the TextInput control. A valid phone number contains at least 10 digits, plus additional formatting characters. For more information, see "Using the PhoneNumberValidator class" on page 1321.

You declare validators in ActionScript either in a script block within an MXML file, or in an ActionScript file, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidatorAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Import PhoneNumberValidator.
            import mx.validators.PhoneNumberValidator;

            // Create the validator.
           private var v:PhoneNumberValidator = new PhoneNumberValidator();

            private function createValidator():void {
                // Configure the validator.
                v.source = phoneInput;
                v.property = "text";
            }
        ]]>
    </mx:Script>

    <!-- Define the TextInput control for entering the phone number. -->
    <mx:TextInput id="phoneInput" creationComplete="createValidator();"/>
    <mx:TextInput id="zipCodeInput"/>
</mx:Application>
```

About the source and property properties

Validators use the following two properties to specify the item to validate:

source Specifies the object containing the property to validate. Set this to an instance of a component or a data model. You use data binding syntax in MXML to specify the value for the `source` property. This property supports dot-delimited Strings for specifying nested properties.

property A String specifying the name of the property of `source` that contains the value to validate.

You can set these properties in any of the following ways:

- In MXML when you use a validator tag.
- In ActionScript by assigning values to the properties.
- When you call the `Validator.validate()` method to invoke a validator programmatically. For more information, see "Triggering validation programmatically" on page 1291.

You often specify a Flex user-interface control as the value of the `source` property, and a property of the control to validate as the value of the `property` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ZCValidator.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:ZipCodeValidator id="zipV"
        source="{myZip}"
        property="text"/>

    <mx:TextInput id="phoneInput"/>
    <mx:TextInput id="myZip"/>
</mx:Application>
```

In this example, you use the Flex ZipCodeValidator to validate the data entered in a TextInput control. The TextInput control stores the input data in its `text` property.

About triggering validation

You trigger validation either automatically in response to an event, or programmatically by an explicit call to the `Validator.validate()` method of a validator.

When you use events, you can cause the validator to execute automatically in response to a user action. For example, you can use the `click` event of a Button control to trigger validation on the fields of a form, or the `valueCommit` event of a TextInput control to trigger validation after a user enters information in the control. For more information, see "Triggering validation by using events" on page 1286.

You can also trigger a validation programmaticaly. For example, you might have to inspect multiple, related input fields to perform a single validation. Or you might have to perform conditional validation based on a user input. For example, you may allow a user to select the currency used for payment, such as U.S. dollars or euros. Therefore, you want to make sure that you invoke a validator configured for the specified currency. In this case, you can make an explicit call to the `validate()` method to trigger the correct validator for the input value. For more information, see "Triggering validation programmatically" on page 1291.

About validating required fields

Flex validators can determine when a user enters an incorrect value into a user-interface control. In addition, all validators support the `required` property, which, if `true`, specifies that a missing or empty value in a user-interface control causes a validation error. The default value is `true`. Therefore, a validation error occurs by default if the user fails to enter any required data in a control associated with a validator. To disable this check, set the `required` property to `false`. For more information, see "Validating required fields" on page 1301.

About validation errors

If a validation error occurs, by default Flex draws a red box around the component associated with the failure. If the user moves the mouse pointer over the component, Flex displays the error message associated with the error. You can customize the look of the component and the error message associated with the error. For more information on validation errors, see "Working with validation errors" on page 1306.

About validation events

Validation is event driven. You can use events to trigger validation, programmatically create and configure validators in response to events, and listen for events dispatched by validators.

For example, when a validation operation completes, a validator dispatches a `valid` or `invalid` event, depending on the results of the validation. You can listen for these events, and then perform any additional processing that your application requires.

Alternatively, Flex components dispatch `valid` and `invalid` events, depending on the results of the validation. This lets you listen for the events being dispatched from the component being validated, rather than listening for events dispatched by the validator.

You are not required to listen for validation events. By default, Flex handles a failed validation by drawing a red box around the control that is associated with the source of the data binding. For a successful validation, Flex clears any indicator of a previous failure. For more information, see "Working with validation events" on page 1310.

About custom validation

Although Flex supplies a number of predefined validators, you might find that you have to implement your own validation logic. The mx.validators.Validator class is an ActionScript class that you can extend to create a subclass that encapsulates your custom validation logic. For more information on creating custom validators, see Chapter 14, "Creating Custom Validators" in *Creating and Extending Flex 2 Components*.

Using validators

This section describes the basic process for using Flex validators. For information on specific validator classes, see "Using standard validators" on page 1313.

Triggering validation by using events

You can trigger validators automatically by associating them with an event. In the following example, the user enters a ZIP code in a TextInput control, and then triggers validation by clicking the Button control:

```
<?xml version="1.0"?>
<!-- validators\ZCValidatorTriggerEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:ZipCodeValidator id="zipV"
        source="{myZip}"
        property="text"
        trigger="{mySubmit}"
        triggerEvent="click"/>

    <mx:TextInput id="myZip"/>
    <mx:Button id="mySubmit" label="Submit"/>
</mx:Application>
```

This example uses the `trigger` and `triggerEvent` properties of the ZipCodeValidator class to associate an event with the validator. These properties have the following values:

trigger Specifies the component generating the event that triggers the validator. If omitted, by default Flex uses the value of the `source` property.

triggerEvent Specifies the event that triggers the validation. If omitted, Flex uses the `valueCommit` event. Flex dispatches the `valueCommit` event whenever the value of a control changes. Usually this is when the user removes focus from the component, or when a property value is changed programmatically. If you want a validator to ignore all events, set `triggerEvent` to an empty string (`""`).

Triggering validation by using the default event

You can rewrite the example from the previous section to use default values for the `trigger` and `triggerEvent` properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ZCValidatorDefEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:ZipCodeValidator id="zipV"
        source="{myZip}"
        property="text"/>

    <mx:TextInput id="myZip"/>
    <mx:Button id="mySubmit" label="Submit"/>
</mx:Application>
```

By omitting the `trigger` and `triggerEvent` properties, Flex triggers the validator when the TextInput control dispatches the `valueCommit` event. Flex controls dispatch the `valueCommit` event when its values changes by user interaction or programmatically.

Triggering validation for data bindings

Data binding provides a syntax for automatically copying the value of a property of one object to a property of another object at run time. With data binding, Flex copies the source value to the destination, typically in response to a modification to the source. The source and destination of data bindings are typically Flex components or data models.

In the next example, you bind data entered in a TextInput control to a data model so that the data in the TextInput control is automatically copied to the data model:

```
<?xml version="1.0"?>
<!-- validators\ValWithDataBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define a data model for storing the phone number. -->
    <mx:Model id="userInfo">
        <phoneInfo>
            <phoneNum>{phoneInput.text}</phoneNum>
        </phoneInfo>
    </mx:Model>

    <!-- Define the TextInput control for entering the phone number. -->
    <mx:TextInput id="phoneInput"/>
    <mx:TextInput id="zipCodeInput"/>
</mx:Application>
```

You can use a validator along with a data binding to validate either the source or destination of the data binding, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerWithDataBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define a data model for storing the phone number. -->
    <mx:Model id="userInfo">
        <phoneInfo>
            <phoneNum>{phoneInput.text}</phoneNum>
        </phoneInfo>
    </mx:Model>

    <!-- Define the PhoneNumberValidator. -->
    <mx:PhoneNumberValidator id="pnV"
        source="{phoneInput}"
        property="text"/>

    <!-- Define the TextInput control for entering the phone number. -->
    <mx:TextInput id="phoneInput"/>
    <mx:TextInput id="zipCodeInput"/>
</mx:Application>
```

This example uses a PhoneNumberValidator to validate the data entered in the TextInput control. In this example, the following occurs:

- You assign the validator to the source of the data binding.
- You use the default event, `valueCommit`, on the TextInput control to trigger the validator. This means the validator executes when the user removes focus from the TextInput control by selecting the TextInput control for the ZIP code.

- Flex updates the destination of the data binding on every change to the source. This means that the `userInfo.phoneNum` field updates on every change to the TextInput control, while the validator executes only when the user removes focus from the TextInput control to trigger the `valueCommit` event. You can use the validator's `triggerEvent` property to specify a different event to trigger the validation.

In a model-view-controller (MVC) design pattern, you isolate the model from the view and controller portions of the application. In the previous example, the data model represents the model, and the TextInput control and validator represents the view.

The TextInput control is not aware that its data is bound to the data model, or that there is any binding on it at all. Since the validator is also assigned to the TextInput control, you have kept the model and view portions of your application separate and can modify one without affecting the other.

However, there is nothing in Flex to prohibit you from assigning a validator to the data model, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerWithDataBindingOnModel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define a data model for storing the phone number. -->
    <mx:Model id="userInfo">
        <phoneInfo>
            <phoneNum>{phoneInput.text}</phoneNum>
        </phoneInfo>
    </mx:Model>

    <!-- Define the PhoneNumberValidator. -->
    <mx:PhoneNumberValidator id="pnV"
        source="{userInfo}"
        property="phoneNum"
        trigger="{phoneInput}"
        listener="{phoneInput}"/>

    <!-- Define the TextInput control for entering the phone number. -->
    <mx:TextInput id="phoneInput"/>
    <mx:TextInput id="zipCodeInput"/>
</mx:Application>
```

In this example, you trigger the data validator using the `valueCommit` event of the TextInput control, but assign the validator to a field of the data model, rather than to a property of the TextInput control.

This example also uses the `listener` property of the validator. This property configures the validator to display validation error information on the specified object, rather than on the source of the validation. In this example, the source of the validation is a model, so you display the visual information on the TextInput control that provided the data to the model. For more information, see "Specifying a listener for validation" on page 1309.

If the model has a nesting structure of elements, you use dot-delimited Strings with the `source` property to specify the model element to validate, as the followoing example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerWithDataBindingComplexModel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define a data model for storing the phone number. -->
    <mx:Model id="userInfo">
        <user>
            <phoneInfo>
                <homePhoneNum>{homePhoneInput.text}</homePhoneNum>
                <cellPhoneNum>{cellPhoneInput.text}</cellPhoneNum>
            </phoneInfo>
        </user>
    </mx:Model>

    <!-- Define the PhoneNumberValidator. -->
    <mx:PhoneNumberValidator id="hPNV"
        source="{userInfo.phoneInfo}"
        property="homePhoneNum"
        trigger="{homePhoneInput}"
        listener="{homePhoneInput}"/>

    <!-- Define the PhoneNumberValidator. -->
    <mx:PhoneNumberValidator id="cPNV"
        source="{userInfo.phoneInfo}"
        property="cellPhoneNum"
        trigger="{cellPhoneInput}"
        listener="{cellPhoneInput}"/>

    <!-- Define the TextInput controls for entering the phone number. -->
    <mx:Label text="Home Phone:"/>
    <mx:TextInput id="homePhoneInput"/>

    <mx:Label text="Cell Phone:"/>
    <mx:TextInput id="cellPhoneInput"/>
</mx:Application>
```

Triggering validation programmatically

The Validator class, and all subclasses of Validator, include a `validate()` method that you can call to invoke a validator directly, rather than triggering the validator automatically using an event.

The `validate()` method has the following signature:

```
validate(value:Object = null,
  supressEvents:Boolean = false):ValidationResultEvent
```

The arguments have the following values:

value If `value` is `null`, use the `source` and `property` properties to specify the data to validate. If `value` is non-null, it specifies a field of an object relative to the `this` keyword, which means an object in the scope of the document.

You should also set the `Validator.listener` property when you specify the `value` argument. When a validation occurs, Flex applies visual changes to the object specified by the `listener` property. By default, Flex sets the `listener` property to the value of the `source` property. However, because you do not specify the `source` property when you pass the `value` argument, you should set it explicitly. For more information, see "Specifying a listener for validation" on page 1309.

supressEvents If `false`, dispatch either the `valid` or `invalid` event on completion. If `true`, do not dispatch events.

This method returns an event object containing the results of the validation that is an instance of the ValidationResultEvent class. For more information on using the return result, see "Handling the return value of the validate() method" on page 1292.

In the following example, you create and invoke a validator programmatically in when a user clicks a Button control:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerProg.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Import ZipCodeValidator.
            import mx.validators.ZipCodeValidator;

            private var v:ZipCodeValidator = new ZipCodeValidator();

            private function performValidation():void {
                v.domain = "US or Canada";
                // Set the listener property to the component
                // used to display validation errors.
                v.listener=myZip;
                v.validate(myZip.text);
            }
        ]]>
    </mx:Script>

    <mx:TextInput id="myZip"/>
    <mx:Button label="Submit" click="performValidation();"/>
</mx:Application>
```

Notice that you are still using an event to trigger the `performValidation()` function that creates and invokes the validator, but the event itself does not automatically invoke the validator.

Any errors in the validator are shown on the associated component, just as if you had triggered the validation directly by using an event.

Handling the return value of the validate() method

You may want to inspect the return value from the `validate()` method to perform some action when the validation succeeds or fails. The `validate()` method returns an event object with a type defined by the ValidationResultEvent class.

The ValidationResultEvent class defines several properties, including the `type` property. The `type` property of the event object contains either `ValidationResultEvent.VALID` or `ValidationResultEvent.INVALID`, based on the validation results. You can use this property as part of your validation logic, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValTriggerProgProcessReturnResult.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Import the ValidationResultEvent class.
            import mx.events.ValidationResultEvent;
            import mx.validators.ZipCodeValidator;

            public var v:ZipCodeValidator = new ZipCodeValidator();

            // Define variable for storing the validation event object.
            public var vResult:ValidationResultEvent;

            public function performValidation():void {
                v.domain = "US or Canada";
                v.listener=myZip;
                vResult = v.validate(myZip.text);

                if (vResult.type==ValidationResultEvent.VALID) {
                    // Validation succeeded.
                    myTA.text='OK';
                }
                else {
                    // Validation failed.
                    myTA.text='Fail';
                }
            }
        ]]>
    </mx:Script>

    <mx:TextInput id="myZip"/>
    <mx:Button label="Submit" click="performValidation();"/>

    <mx:TextArea id="myTA"/>
</mx:Application>
```

The ValidationResultEvent class has additional properties that you can use when processing validation events. For more information, see "Working with validation events" on page 1310.

Triggering the DateValidator and CreditCardValidator

The DateValidator and CreditCardValidator can validate multiple fields by using a single validator. A CreditCardValidator examines one field that contains the credit card number and a second field that contains the credit card type. The DateValidator can examine a single field that contains a date, or multiple fields that together make up a date.

When you validate an object that contains multiple properties that are set independently, you often cannot use events to automatically trigger the validator because no single field contains all of the information required to perform the validation.

One way to validate a complex object to call the `validate()` method of the validator based on some type of user interaction. For example, you might want to validate the multiple fields that make up a date in response to the `click` event of a Button control, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\DateAndCC.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define the data model. -->
    <mx:Model id="date">
        <dateInfo>
            <month>{monthInput.text}</month>
            <day>{dayInput.text}</day>
            <year>{yearInput.text}</year>
        </dateInfo>
    </mx:Model>

    <!-- Define the validators. -->
    <mx:DateValidator id="dayV"
        triggerEvent=""
        daySource="{dayInput}"
        dayProperty="text"
        monthSource="{monthInput}"
        monthProperty="text"
        yearSource="{yearInput}"
        yearProperty="text"/>

    <!-- Define the form to populate the model. -->
    <mx:Form>
        <mx:TextInput id="monthInput"/>
        <mx:TextInput id="dayInput"/>
        <mx:TextInput id="yearInput"/>
    </mx:Form>

    <!-- Define the button to trigger validation. -->
    <mx:Button label="Submit"
        click="dayV.validate();"/>
```

```
</mx:Application>
```

The validator in this example examines all three input fields. If any field is invalid, validation fails. The validator highlights only the invalid fields that failed. For more information on how validators signal validation errors, see "Working with validation errors" on page 1306. For more information on the DateValidator and CreditCardValidator, see "Using standard validators" on page 1313.

Invoking multiple validators in a function

You can invoke multiple validators programmatically from a single function. In this example, you use the ZipCodeValidator and PhoneNumberValidator validators to validate the ZIP code and phone number input to a data model.

```
<?xml version="1.0"?>
<!-- validators\ValidatorCustomFuncStaticVals.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Import event class.
            import mx.events.ValidationResultEvent;

            // Define variable for storing the validation event object.
            private var vResult:ValidationResultEvent;

            private function validateZipPhone():void {
                // Validate the ZIP code.
                vResult = zipV.validate();
                // If the ZIP code is invalid,
                // do not move on to the next field.
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

                // Validate the phone number.
                vResult = pnV.validate();
                // If the phone number is invalid,
                // do not move on to the validation.
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;
            }
        ]]>
    </mx:Script>

    <mx:Model id="person">
        <userInfo>
            <zipCode>{zipCodeInput.text}</zipCode>
            <phoneNumber>{phoneNumberInput.text}</phoneNumber>
```

```
        </userInfo>
    </mx:Model>

    <!-- Define the validators. -->
    <mx:ZipCodeValidator id="zipV"
        source="{zipCodeInput}"
        property="text"/>
    <mx:PhoneNumberValidator id="pnV"
        source ="{phoneNumberInput}"
        property="text"/>

    <mx:Form>
        <!-- Collect input data -->
        <mx:TextInput id="zipCodeInput"/>
        <mx:TextInput id="phoneNumberInput"/>
    </mx:Form>

    <mx:Button label="Validate"
        click="validateZipPhone();"/>
</mx:Application>
```

In this example, you use the predefined ZipCodeValidator and PhoneNumberValidator to validate user information as the user enters it. Then, when the user clicks the Submit button to submit the form, you validate that the ZIP code is actually within the specified area code of the phone number.

You can also use the static `Validator.validateAll()` method to invoke all of the validators in an Array. This method returns an Array containing one ValidationResultEvent object for each validator that failed, and an empty Array if all validators succeed. The following example uses this method to invoke two validators in response to the `click` event for the Button control:

```
<?xml version="1.0"?>
<!-- validators\ValidatorMultipleValids.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initValidatorArray();">

    <mx:Script>
        <![CDATA[
            import mx.validators.Validator;

            // Define the validator Array.
            private var myValidators:Array;

            private function initValidatorArray():void {
                myValidators=[zipV, pnV];
            }
        ]]>
    </mx:Script>

    <mx:Model id="person">
        <userInfo>
            <zipCode>{zipCodeInput.text}</zipCode>
            <phoneNumber>{phoneNumberInput.text}</phoneNumber>
        </userInfo>
    </mx:Model>

    <!-- Define the validators. -->
    <mx:ZipCodeValidator id="zipV"
        source="{zipCodeInput}"
        property="text"/>
    <mx:PhoneNumberValidator id="pnV"
        source ="{phoneNumberInput}"
        property="text"/>

    <mx:Form>
        <!-- Collect input data -->
        <mx:TextInput id="zipCodeInput"/>
        <mx:TextInput id="phoneNumberInput"/>
    </mx:Form>

    <mx:Button label="Validate"
        click="Validator.validateAll(myValidators);"/>
</mx:Application>
```

Creating a reusable validator

You can define a reusable validator so that you can use it to validate multiple fields. To make it reusable, you programmatically set the `source` and `property` properties to specify the field to validate, or pass that information to the `validate()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ReusableVals.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private function performValidation(eventObj:Event):void {
                zipV.listener=eventObj.currentTarget;
                zipV.validate(eventObj.currentTarget.text);
            }
        ]]>
    </mx:Script>

    <mx:ZipCodeValidator id="zipV"
        triggerEvent=""/>

    <mx:TextInput id="shippingZip"
        focusOut="performValidation(event);"/>
    <mx:TextInput id="billingZip"
        focusOut="performValidation(event);"/>
</mx:Application>
```

In this example, you have two address areas for a customer: one for a billing address and one for a shipping address. Both addresses have a ZIP code field, so you can reuse a single ZipCodeValidator for both fields. The event listener for the `focusOut` event passes the field to validate to the `validate()` method.

Alternatively, you can write the `performValidation()` function as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ReusableValsSpecifySource.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private function performValidation(eventObj:Event):void {
                zipV.source = eventObj.currentTarget;
                zipV.property = "text";
                zipV.validate();
            }
        ]]>
    </mx:Script>

    <mx:ZipCodeValidator id="zipV"
        triggerEvent=""/>

    <mx:TextInput id="shippingZip"
        focusOut="performValidation(event);"/>
    <mx:TextInput id="billingZip"
        focusOut="performValidation(event);"/>
</mx:Application>
```

Conditionalizing validator execution

By invoking a validator programmatically, you can use conditional logic in your application to determine which of several validators to invoke, set validator properties, or perform other preprocessing or postprocessing as part of the validation.

In the next example, you use a Button control to invoke a validator, but the application first determines which validator to execute, based on user input:

```
<?xml version="1.0"?>
<!-- validators\ConditionalVal.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.events.ValidationResultEvent;

            private var vEvent:ValidationResultEvent;

            private function validateData():void {
                if (country.selectedValue == "Canada") {
                    vEvent = zipCN.validate(zipInput.text);
                }
                else            {
                    vEvent = zipUS.validate(zipInput.text);
                }
            }
        ]]>
    </mx:Script>

    <mx:ZipCodeValidator id="zipUS"
        domain="US Only"
        listener="{zipInput}"/>
    <mx:ZipCodeValidator id="zipCN"
        domain="US or Canada"
        listener="{zipInput}"/>

    <mx:RadioButtonGroup id="country"/>
    <mx:RadioButton groupName="country" label="US"/>
    <mx:RadioButton groupName="country" label="Canada"/>

    <mx:TextInput id="zipInput"/>

    <mx:Button label="Submit" click="validateData();"/>
</mx:Application>
```

In this example, you use a ZipCodeValidator to validate a ZIP code entered into a TextInput control. However, the `validateData()` function must first determine whether the ZIP code is for the U.S. or for Canada before performing the validation. In this example, the application uses the RadioButton controls to let the user specify the country as part of entering the ZIP code.

Validating required fields

The Validator class, the base class for all Flex validators, contains a `required` property that when set to `true` causes validation to fail when a field is empty. You use this property to configure the validator to fail when a user does not enter data in a required input field.

You typically call the `validate()` method to invoke a validator on a required field. This is often necessary because you cannot guarantee that an event occurs to trigger the validation—an empty input field often means that the user never gave the input control focus.

The following example performs a validation on a required input field:

```
<?xml version="1.0"?>
<!-- validators\RequiredVal.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:StringValidator id="reqV"
        source="{inputA}"
        property="text"
        required="true"/>

    <mx:TextInput id="inputA"/>

    <mx:Button label="Submit"
        click="reqV.validate();"/>
</mx:Application>
```

In this example, the StringValidator executes when the following occurs:

- The TextInput control dispatches the `valueCommit` event. However, to dispatch that event, the user must give the TextInput control focus, and then remove focus. If the user never gives the TextInput control focus, the validator does not trigger, and Flex does not recognize that the control is empty. Therefore, you must call the `validate()` method to ensure that the validator checks for missing data.
- The user clicks the Button control. The validator issues a validation error when the user does not enter any data into the TextInput control. It also issues a validation error if the user enters an invalid String value.

Enabling and disabling a validator

The `Validator.enabled` property lets you enable and disable a validator. When the value of the `enabled` property is `true`, the validator is enabled; when the value is `false`, the validator is disabled. When a validator is disabled, it dispatches no events, and the `validate()` method returns `null`.

For example, you can set the `enabled` property using data binding, as the following code shows:

```
<?xml version="1.0"?>
<!-- validators\EnableVal.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:ZipCodeValidator id="zcVal"
        source="{inputA}"
        property="text"
        required="true"
        enabled="{enableV.selected}"/>

    <mx:TextInput id="inputA"/>
    <mx:TextInput/>
    <mx:CheckBox id="enableV"
        label="Validate input?"/>
</mx:Application>
```

In this example, you enable the validator only when the user selects the CheckBox control.

Using data binding to configure validators

You configure validators to match your application requirements. For example, the StringValidator lets you specify a minimum and maximum length of a valid string. For a String to be considered valid, it must be at least the minimum number of characters long, and less than or equal to the maximum number of characters.

Often, you set validator properties statically, which means that they do not change as your application executes. For example, the following StringValidator defines that the input string must be at least one character long and no longer than 10 characters:

```
<mx:StringValidator required="true" minlength="1" maxLength="10"/>
```

User input might also define the properties of the validator. In the following example, you let the user set the minimum and maximum values of a NumberValidator:

```
<?xml version="1.0"?>
<!-- validators\ConfigWithBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:NumberValidator
        source="{inputA}"
        property="text"
        minValue="{Number(inputMin.text)}"
        maxValue="{Number(inputMax.text)}"/>

    <mx:TextInput id="inputA"/>
    <mx:TextInput id="inputMin" text="1"/>
    <mx:TextInput id="inputMax" text="10"/>
</mx:Application>
```

In this example, you use data binding to configure the properties of the validators.

General guidelines for validation

You should be aware of some guidelines when performing validation on forms. Typically, you associate forms with data models. That lets you trigger validation as part of binding an input user interface control to a field of the data model. You can also perform some of the following actions:

- If possible, assign validators to all the individual user-interface controls of the form. You can use a validator even if all that you want to do is to ensure that the user entered a value.
- Assign validators to multiple fields when necessary. For example, use the CreditCardValidator or the DateValidator with multiple fields.
- If you have any required fields, ensure that you explicitly call the `validate()` method on the validator. For more information, see "Validating required fields" on page 1301.
- Define a Submit button to invoke any validators before submitting data to a server. Typically, you use the `click` event of the Button control to invoke validators programmatically, and then submit the data if all validation succeeds.

The following example uses many of these guidelines to validate a form made up of several TextInput controls:

```
<?xml version="1.0"?>
<!-- validators\FullApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.events.ValidationResultEvent;

            private var vResult:ValidationResultEvent;

            // Function to validate data and submit it to the server.
            private function validateAndSubmit():void           {
                // Validate the required fields.
                vResult = fNameV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

                vResult = lNameV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

               // Since the date requires 3 fields, perform the validation
                // when the Submit button is clicked.
                vResult = dayV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;

                // Invoke any other validators or validation logic to make
                // an additional check before submitting the data.

                // Submit data to server.
            }
        ]]>
    </mx:Script>

    <!-- Define the data model. -->
    <mx:Model id="formInfo">
        <formData>
            <date>
                <month>{monthInput.text}</month>
                <day>{dayInput.text}</day>
                <year>{yearInput.text}</year>
            </date>
            <name>
                <firstName>{fNameInput.text}</firstName>
                <lastName>{lNameInput.text}</lastName>
            </name>
```

```
            <phoneNum>{phoneInput.text}</phoneNum>
        </formData>
    </mx:Model>

    <!-- Define the validators. -->
    <mx:StringValidator id="fNameV"
        required="true"
        source="{fNameInput}"
        property="text"/>
    <mx:StringValidator id="lNameV"
        required="true"
        source="{lNameInput}"
        property="text"/>
    <mx:PhoneNumberValidator id="pnV"
        source="{phoneInput}"
        property="text"/>

    <!-- Invoke the DataValidator programmatically. -->
    <mx:DateValidator id="dayV"
        triggerEvent=""
        daySource="{dayInput}" dayProperty="text"
        monthSource="{monthInput}" monthProperty="text"
        yearSource="{yearInput}" yearProperty="text"/>

    <!-- Define the form to populate the model. -->
    <mx:Form>
        <mx:FormItem label="Month">
            <mx:TextInput id="monthInput"/>
        </mx:FormItem>
        <mx:FormItem label="Day">
            <mx:TextInput id="dayInput"/>
        </mx:FormItem>
        <mx:FormItem label="Year">
            <mx:TextInput id="yearInput"/>
        </mx:FormItem>
        <mx:FormItem label="First name">
            <mx:TextInput id="fNameInput"/>
        </mx:FormItem>
        <mx:FormItem label="Last name">
            <mx:TextInput id="lNameInput"/>
        </mx:FormItem>
        <mx:FormItem label="Phone">
            <mx:TextInput id="phoneInput"/>
        </mx:FormItem>
    </mx:Form>

    <!-- Define the button to trigger validation. -->
    <mx:Button label="Submit"
        click="validateAndSubmit();"/>
</mx:Application>
```

In this example the following actions occur:

- The associated validator executes whenever the TextInput control dispatches a `valueCommit` event.
- The `click` event for the Button control invokes the `validateAndSubmit()` function to perform final validation before submitting data to the server.
- The `validateAndSubmit()` function invokes the validators for all required fields.
- The `validateAndSubmit()` function invokes the DateValidator because it requires three different input fields.
- Upon detecting the first validation error, the `validateAndSubmit()` function returns but does not submit the data.
- When all validations succeed, s the `validateAndSubmit()` function submits the data to the server.

Working with validation errors

Subclasses of the UIComponent base class, which include the Flex user-interface components, generally handle validation failures by changing their border color and displaying an error message. When validation succeeds, components hide any existing validation error message and remove any border.

This section describes how to configure the content of the error messages and the display characteristics of validation errors.

Configuring error messages

All Flex validators define default error messages. In most cases, you can override these messages with your own.

The default error messages for all validators are defined using resource bundles so that you can easily change them as part of localizing your application. You can override the default value of an error message for all validator objects created from a validator class by editing the resource bundles associated with that class.

You edit the error message for a specific validator object by writing a String value to a property of the validator. For example, the PhoneNumberValidator defines a default error message to indicate that an input phone number has the wrong number of digits. You can override the default error message for a specific PhoneNumberValidator object by assigning a new message string to the `wrongLengthError` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidatorErrMessage.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Define the PhoneNumberValidator. -->
    <mx:PhoneNumberValidator id="pnV"
        source="{phoneInput}" property="text"
        wrongLengthError="Please enter a 10-digit number."/>

    <!-- Define the TextInput control for entering the phone number. -->
    <mx:TextInput id="phoneInput"/>
    <mx:TextInput id="zipCodeInput"/>
</mx:Application>
```

Changing the color of the validation error message

By default, the validation error message that appears when you move the mouse pointer over a user-interface control has a red background. You can use the `ErrorTip` style to change the color, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\PNValidatorErrMessageStyle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Use blue for the error message. -->
    <mx:Style>
        .errorTip { borderColor: #0000FF}
    </mx:Style>

    <!-- Define the PhoneNumberValidator. -->
    <mx:PhoneNumberValidator id="pnV"
        source="{phoneInput}" property="text"
        wrongLengthError="Please enter a 10-digit number."/>

    <!-- Define the TextInput control for entering the phone number. -->
    <mx:TextInput id="phoneInput"/>
    <mx:TextInput id="zipCodeInput"/>
</mx:Application>
```

In this example, the error message appears as light green.

Showing a validation error by using errorString

The UIComponent class defines the `errorString` property that you can use to show a validation error for a component, without actually using a validator class. When you write a String value to the `UIComponent.errorString` property, Flex draws a red border around the component to indicate the validation error, and the String appears in a ToolTip as the validation error message when you move the mouse over the component, just as if a validator detected a validation error.

To clear the validation error, write an empty String, " ", to the `UIComponent.errorString` property.

NOTE

Writing a value to the `UIComponent.errorString` property does not trigger the `valid` or `invalid` events; it only changes the border color and displays the validation error message.

For information on writing custom ToolTip controls, see Chapter 23, "Using ToolTips," on page 943.

Clearing a validation error

The `errorString` property is useful when you want to reset a field that is a source for validation, and prevent a validation error from occurring when you reset the field.

For example, you might provide a form to gather user input. Within your form, you might also provide a button, or other mechanism, that lets the user reset the form. However, clearing form fields that are tied to validators could trigger a validation error. The following example uses the `errorString` property as part of resetting the `text` property of a TextInput control to prevent validation errors when the form resets:

```
<?xml version="1.0"?>
<!-- validators\ResetVal.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.events.ValidationResultEvent;
            private var vResult:ValidationResultEvent;

            // Function to validate data and submit it to the server.
            private function validateAndSubmit():void {
                // Validate the required fields.
                vResult = zipV.validate();
                if (vResult.type==ValidationResultEvent.INVALID)
                    return;
```

```
                // Submit data to server.
            }

            <!-- Clear the input controls and the errorString property
                when resetting the form. -->
            private function resetForm():void {
                zipInput.text = '';
                zipInput.errorString = '';
            }
        ]]>
    </mx:Script>

    <mx:ZipCodeValidator id="zipV"
        source="{zipInput}"
        property="text"/>

    <mx:Form>
        <mx:FormItem label="Enter ZIP code">
            <mx:TextInput id="zipInput"/>
        </mx:FormItem>
        <mx:FormItem label="Enter Country">
            <mx:TextInput id="cntryInput"/>
        </mx:FormItem>
    </mx:Form>

    <!-- Trigger submit. -->
    <mx:Button label="Submit" click="validateAndSubmit();"/>

    <!-- Trigger reset. -->
    <mx:Button label="Reset" click="resetForm();"/>
</mx:Application>
```

In this example, the function that clears the form items also clears the `errorString` property associated with each item, clearing any validation errors.

Specifying a listener for validation

All validators support a `listener` property. When a validation occurs, Flex applies visual changes to the object specified by the `listener` property.

By default, Flex sets the `listener` property to the value of the `source` property. That means that all visual changes that occur to reflect a validation event occur on the component being validated. However, you might want to validate one component but have validation results apply to a different component, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\SetListener.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:ZipCodeValidator id="zipV"
        source="{zipCodeInput}"
        property="text"
        listener="{errorMsg}"/>

    <mx:TextInput id="zipCodeInput"/>
    <mx:TextArea id="errorMsg"/>
</mx:Application>
```

Working with validation events

Flex gives you two ways to listen for validation events:

- Listen for validation events dispatched by the component being validated.

  Flex components dispatch `valid` and `invalid` events, depending on the results of the validation. This lets you listen for the events being dispatched from the component being validated, and perform any additional processing on the component based on its validation result.

  The event object passed to the event listener is of type Event. For more information, including an example, see "Explicitly handling component validation events" on page 1311.

- Listen for validation events dispatched by validators.

  All validators dispatch `valid` or `invalid` events, depending on the results of the validation. You can listen for these events, and then perform any additional processing as required by your validator.

  The event object passed to the event listener is of type ValidationResultEvent. For more information, including an example, see "Explicitly handing validator validation events" on page 1312.

You are not required to listen for validation events. When these events occur, by default, Flex changes the appropriate border color of the target component, displays an error message for an `invalid` event, or hides any previous error message for a `valid` event.

Explicitly handling component validation events

Sometimes you might want to perform some additional processing for a component if a validation fails or succeeds. In that case, you can handle the `valid` and `invalid` events yourself. The following example defines a event listener for the `invalid` event to perform additional processing when a validation fails:

```
<?xml version="1.0"?>
<!-- validators\ValCustomEventListener -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Import event class.
            import flash.events.Event;

            // Define vars for storing text colors.
            private var errorTextColor:Object = "red";
            private var currentTextColor:Object;

          // Initialization event handler for getting default text color.
         private function myCreationComplete(eventObj:Event):void {
                currentTextColor = getStyle('color');
            }

            // For an invalid event, change the text color.
            private function handleInvalidVal(eventObject:Event):void {
                setStyle('color', errorTextColor);
            }

            // For a valid event, restore the text color.
            private function handleValidVal(eventObject:Event):void {
                setStyle('color', currentTextColor);
            }
        ]]>
    </mx:Script>

    <mx:PhoneNumberValidator source="{phoneInput}" property="text"/>

    <mx:TextInput id="phoneInput"
        initialize="myCreationComplete(event);"
        invalid="handleInvalidVal(event);"
        valid="handleValidVal(event);"/>
    <mx:TextInput id="zipInput"/>
</mx:Application>
```

Explicitly handing validator validation events

To explicitly handle the `valid` and `invalid` events dispatched by validators, define an event listener, as the following example shows:

```
<?xml version="1.0"?>
<!-- validators\ValEventListener.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Import event class
            import mx.events.ValidationResultEvent;

           private function handleValid(event:ValidationResultEvent):void {
                if(event.type==ValidationResultEvent.VALID)
                    submitButton.enabled = true;
                else
                    submitButton.enabled = false;
            }

            // Submit form is everything is valid.
            private function submitForm():void {
                // Handle submit.
            }

        ]]>
    </mx:Script>

    <mx:ZipCodeValidator
        source="{inputZip}" property="text"
        valid="handleValid(event);"
        invalid="handleValid(event);"/>

    <mx:TextInput id="inputZip"/>
    <mx:TextInput id="inputPn"/>

    <mx:Button id="submitButton"
        label="Submit"
        enabled="false"
        click="submitForm();"/>
</mx:Application>
```

In this example, the Button control is disabled until the TextInput field contains a valid ZIP code. The `type` property of the event object is either `ValidationResultEvent.VALID` or `ValidationResultEvent.INVALID`, based on the result of the validation.

Within the event listener, you can use all the properties of the ValidationResultEvent class, including the following:

field A String that contains the name of the field that failed validation and triggered the event.

message A String that contains all the validator error messages created by the validation.

results An Array of ValidationResult objects, one for each field examined by the validator. For a successful validation, the `ValidationResultEvent.results` Array property is empty. For a validation failure, the `ValidationResultEvent.results` Array property contains one ValidationResult object for each field checked by the validator, both for fields that failed the validation and for fields that passed. Examine the `ValidationResult.isError` property to determine if the field passed or failed the validation.

Using standard validators

Flex includes the Validator subclasses. You use these validators for common types of data, including credit card numbers, dates, e-mail addresses, numbers, phone numbers, Social Security numbers, strings, and ZIP codes. This section describes the following validators:

- Using the CreditCardValidator class
- Using the CurrencyValidator class
- Using the DateValidator class
- Using the EmailValidator class
- Using the NumberValidator class
- Using the PhoneNumberValidator class
- Using the RegExpValidator class
- Using the SocialSecurityValidator class
- Using the StringValidator class
- Using the ZipCodeValidator class

Using the CreditCardValidator class

The CreditCardValidator class validates that a credit card number is the correct length, has the correct prefix, and passes the Luhn mod10 algorithm for the specified card type. This validator does not check whether the credit card is an actual active credit card account.

You typically use the `cardNumberSource` and `cardNumberProperty` properties to specify the location of the credit card number, and the `cardTypeSource` and `cardTypeProperty` properties to specify the location of the credit card type to validate.

The CreditCardValidator class validates that a credit card number is the correct length for the specified card type, as follows:

- Visa: 13 or 16 digits
- MasterCard: 16 digits
- Discover: 16 digits
- American Express: 15 digits
- DinersClub: 14 digits, or 16 digits if it also functions as a MasterCard

You specify the type of credit card number to validate by assigning a constant to the `cardTypeProperty` property. In MXML, valid constants values are:

- `"American Express"`
- `"Diners Club"`
- `"Discover"`
- `"MasterCard"`
- `"Visa"`

In ActionScript, you can use the following constants to set the `cardTypeProperty` property:

- `CreditCardValidatorCardType.AMERICAN_EXPRESS`
- `CreditCardValidatorCardType.DINERS_CLUB`
- `CreditCardValidatorCardType.DISCOVER`
- `CreditCardValidatorCardType.MASTERCARD`
- `CreditCardValidatorCardType.VISA`

The following example validates a credit card number based on the card type that the users specifies. Any validation errors propagate to the Application object and open an Alert window.

```
<?xml version="1.0"?>
<!-- validators\CCExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:CreditCardValidator id="ccV"
        cardTypeSource="{cardTypeCombo.selectedItem}"
        cardTypeProperty="data"
        cardNumberSource="{cardNumberInput}"
        cardNumberProperty="text"/>

    <mx:Form id="creditCardForm">
        <mx:FormItem label="Card Type">
            <mx:ComboBox id="cardTypeCombo">
                <mx:dataProvider>
                    <mx:Object label="American Express"
                        data="American Express"/>
                    <mx:Object label="Diners Club"
                        data="Diners Club"/>
                    <mx:Object label="Discover"
                        data="Discover"/>
                    <mx:Object label="MasterCard"
                        data="MasterCard"/>
                    <mx:Object label="Visa"
                        data="Visa"/>
                </mx:dataProvider>
            </mx:ComboBox>
        </mx:FormItem>
        <mx:FormItem label="Credit Card Number">
            <mx:TextInput id="cardNumberInput"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button label="Check Credit" click="ccV.validate();"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Using the CurrencyValidator class

The CurrencyValidator class checks that a string is a valid currency expression based on a set of parameters. The CurrencyValidator class defines the properties that let you specify the format of the currency value, whether to allow negative values, and the precision of the values.

The following example uses the CurrencyValidator class to validate a currency value entered in U.S. dollars and in euros:

```
<?xml version="1.0"?>
<!-- validators\CurrencyExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Example for US currency. -->
    <mx:CurrencyValidator id="usV"
        source="{priceUS}" property="text"
        alignSymbol="left"
        trigger="{valButton}"
        triggerEvent="click"/>

    <mx:Label text="Enter a US-formatted price:"/>
    <mx:TextInput id="priceUS"/>

    <!-- Example for European currency. -->
    <mx:CurrencyValidator id="eurV"
        source="{priceEU}" property="text"
        alignSymbol="right"
        decimalSeparator="," thousandsSeparator="."
        trigger="{valButton}"
        triggerEvent="click"/>

    <mx:Label text="Enter a European-formatted price:"/>
    <mx:TextInput id="priceEU"/>

    <mx:Button id="valButton" label="Validate Currencies"/>
</mx:Application>
```

Using the DateValidator class

The DateValidator class validates that a String or Object contains a proper date and matches a specified format. Users can enter a single digit or two digits for month, day, and year. By default, the validator ensures that the following information is provided:

- The month is between 1 and 12
- The day is between 1 and 31
- The year is a number

If you specify a single String to validate, the String can contain digits and the formatting characters that the `allowedFormatChars` property specifies, including the slash (/), backslash (\), dash (-), and period (.) characters. By default, the input format of the date in a String is "mm/dd/yyyy" where "mm" is the month, "dd" is the day, and "yyyy" is the year. You can use the `inputFormat` property to specify a different format.

You can also specify to validate a date represented by a single Object, or by multiple fields of different objects. For example, you could use a data model that contains three fields that represent the day, month, and year portions of a date, or three TextInput controls that let a user enter a date as three separate fields. Even if you specify a date format that excludes a day, month, or year element, you must specify all three fields to the validator.

The following table describes how to specify the date to the DateValidator:

Validation source	Required properties	Default listener
Single String containing the date	Use the `source` and `property` properties to specify the String.	Flex associates error messages with the field specified by the `property` property.
Object or multiple fields containing the day, month, and year	Use all of the following properties to specify the day, month, and year inputs: `daySource`, `dayProperty`, `monthSource`, `monthProperty`, `yearSource`, and `yearProperty`.	Flex associates error messages with the field specified by the `daySource`, `monthSource`, and `yearSource` properties, depending on the field that caused the validation error.

The following example validates a date entered into a form:

```
<?xml version="1.0"?>
<!-- validators\DateExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:DateValidator id="dateV"
        daySource="{dayInput}" dayProperty="text"
        monthSource="{monthInput}" monthProperty="text"
        yearSource="{yearInput}" yearProperty="text"/>

    <mx:Form >
        <mx:FormItem label="Month">
            <mx:TextInput id="monthInput"/>
        </mx:FormItem>
        <mx:FormItem label="Day">
            <mx:TextInput id="dayInput"/>
        </mx:FormItem>
        <mx:FormItem label="Year">
            <mx:TextInput id="yearInput"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button label="Check Date" click="dateV.validate();"/>
        </mx:FormItem>
    </mx:Form>

    <!-- Alternate method for a single field containing the date. -->
    <mx:Model id="alternateDate">
        <dateInfo>
            <date>{dateInput.text}</date>
        </dateInfo>
    </mx:Model>

    <mx:DateValidator id="stringDateV"
        source="{dateInput}" property="text"
        inputFormat="dd/mm/yyyy"
        allowedFormatChars="*#~/"/>

    <mx:Form>
        <mx:FormItem label="Date of Birth (dd/mm/yyyy)">
            <mx:TextInput id="dateInput"/>
        </mx:FormItem>
        <mx:FormItem>
          <mx:Button label="Check Date" click="stringDateV.validate();"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Using the EmailValidator class

The EmailValidator class validates that a string has an at sign character (@) and a period character (.) in the domain. You can use IP domain names if they are enclosed in square brackets; for example, myname@[206.132.22.1]. You can use individual IP numbers from 0 to 255. This validator does not check whether the domain and user name actually exist.

The following example validates an e-mail address to ensure that it is formatted correctly:

```
<?xml version="1.0"?>
<!-- validators\EmailExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form id="contactForm">
        <mx:FormItem id="homePhoneItem" label="Home Phone">
            <mx:TextInput id="homePhoneInput"/>
        </mx:FormItem>
        <mx:FormItem id="cellPhoneItem" label="Cell Phone">
            <mx:TextInput id="cellPhoneInput"/>
        </mx:FormItem>
        <mx:FormItem id="emailItem" label="Email">
            <mx:TextInput id="emailInput"/>
        </mx:FormItem>
    </mx:Form>

    <mx:PhoneNumberValidator id="pnVHome"
        source="{homePhoneInput}" property="text"/>
    <mx:PhoneNumberValidator id="pnVCell"
        source="{cellPhoneInput}" property="text"/>
    <mx:EmailValidator id="emV"
        source="{emailInput}" property="text"/>
</mx:Application>
```

Using the NumberValidator class

The NumberValidator class ensures that a string represents a valid number. This validator can ensure that the input falls within a given range (specified by the `minValue` and `maxValue` properties), is an integer (specified by the `domain` property), is non-negative (specified by the `allowNegative` property), and does not exceed the specified precision. The NumberValidator correctly validates formatted numbers (for example, "12,345.67"), and you can customize its `thousandsSeparator` and `decimalSeparator` properties for internationalization.

The following example uses the NumberValidator class to ensure that an integer is between 1 and 10:

```
<?xml version="1.0"?>
<!-- validators\NumberExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form >
        <mx:FormItem
            label="Number of Widgets (max 10 per customer)">
            <mx:TextInput id="quantityInput"/>
        </mx:FormItem>
        <mx:FormItem >
            <mx:Button label="Submit"/>
        </mx:FormItem>
    </mx:Form>

    <mx:NumberValidator id="numV"
        source="{quantityInput}" property="text"
        minValue="1" maxValue="10" domain="int"/>
</mx:Application>
```

Using the PhoneNumberValidator class

The PhoneNumberValidator class validates that a string is a valid phone number. A valid phone number contains at least 10 digits, plus additional formatting characters. This validator does not check if the phone number is an actual active phone number.

The following example uses two PhoneNumberValidator tags to ensure that the home and mobile phone numbers are entered correctly:

```
<?xml version="1.0"?>
<!-- validators\PhoneExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form id="contactForm">
        <mx:FormItem id="homePhoneItem" label="Home Phone">
            <mx:TextInput id="homePhoneInput"/>
        </mx:FormItem>
        <mx:FormItem id="cellPhoneItem" label="Cell Phone">
            <mx:TextInput id="cellPhoneInput"/>
        </mx:FormItem>
        <mx:FormItem id="emailItem" label="Email">
            <mx:TextInput id="emailInput"/>
        </mx:FormItem>
    </mx:Form>

    <mx:PhoneNumberValidator id="pnVHome"
        source="{homePhoneInput}" property="text"/>
    <mx:PhoneNumberValidator id="pnVCell"
        source="{cellPhoneInput}" property="text"/>
    <mx:EmailValidator id="emV"
        source="{emailInput}" property="text"/>
</mx:Application>
```

Using the RegExpValidator class

The RegExpValidator class lets you use a regular expression to validate a field. You pass a regular expression to the validator by using the `expression` property, and additional flags to control the regular expression pattern matching by using the `flags` property.

The validation is successful if the validator can find a match of the regular expression in the field to validate. A validation error occurs when the validator finds no match.

This section describes how to use regular expressions with the RegExpValidator. For information on writing regular expressions, see *Programming ActionScript 3.0*.

The RegExpValidator class dispatches the `valid` and `invalid` events. For an `invalid` event, the event object is an instance of the ValidationResultEvent class, and it contains an Array of ValidationResult objects.

However, for a `valid` event, the ValidationResultEvent object contains an Array of RegExpValidationResult objects. The RegExpValidationResult class is a child class of the ValidationResult class, and contains additional properties that you use with regular expressions, including the following:

matchedIndex An integer that contains the starting index in the input String of the match.

matchedString A String that contains the substring of the input String that matches the regular expression.

matchedSubStrings An Array of Strings that contains parenthesized substring matches, if any. If no substring matches are found, this Array is of length 0. Use `matchedSubStrings[0]` to access the first substring match.

The following example uses the regular expression ABC\d to cause the validator to match a pattern consisting of the letters A, B, and C in sequence followed by any digit:

```
<?xml version="1.0"?>
<!-- validators\RegExpExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
          import mx.events.ValidationResultEvent;
          import mx.validators.*;

          private function handleResult(event:ValidationResultEvent):void {
              if (event.type == "valid")
              {
                  // For valid events, the results Array contains
                  // RegExpValidationResult objects.
                  var xResult:RegExpValidationResult;
                  myTA.text="";
                  for (var i:uint = 0; i < event.results.length; i++)
                  {
                      xResult = event.results[i];
                      myTA.text=myTA.text + xResult.matchedIndex + " " +
                          xResult.matchedString + "\n";
                  }
              }
              else
              {
                  // Not necessary, but if you needed to access it,
                  // the results array contains ValidationResult objects.
                  var result:ValidationResult;
                  myTA.text="";
              }
          }
        ]]>
    </mx:Script>
```

```
    <mx:RegExpValidator id="regExpV"
        source="{exp}" property="text"
        flags="g"
        expression="{source.text}"
        valid="handleResult(event);"
        invalid="handleResult(event);"/>

    <mx:Form>
        <mx:FormItem label="Search string">
            <mx:TextInput id="exp"/>
        </mx:FormItem>
        <mx:FormItem label="Regular expression">
            <mx:TextInput id="source" text="ABC\d"/>
        </mx:FormItem>
        <mx:FormItem label="Results">
            <mx:TextArea id="myTA"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

In this example, you specify the regular expression in the TextInput control named source, and bind it to the `expression` property of the validator. You can modify the regular expression by entering a new expression in the TextInput control. A value of `g` for the `flags` property specifies to find multiple matches in the input field.

The event handler for the `valid` event writes to the TextArea control the index in the input String and matching substring of all matches of the regular expression. The `invalid` event handler clears the TextArea control.

Using the SocialSecurityValidator class

The SocialSecurityValidator class validates that a string is a valid United States Social Security Number. This validator does not check if the number is an existing Social Security Number.

The following example validates a Social Security Number:

```
<?xml version="1.0"?>
<!-- validators\SSExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form id="identityForm">
        <mx:FormItem id="ssnItem" label="Social Security Number">
            <mx:TextInput id="ssnField"/>
        </mx:FormItem>
        <mx:FormItem id="licenseItem" label="Driver's License Number">
            <mx:TextInput id="licenseInput"/> <!-- Not validated -->
        </mx:FormItem>
    </mx:Form>

    <mx:SocialSecurityValidator id="ssV"
        source="{ssnField}" property="text"/>
</mx:Application>
```

Using the StringValidator class

The StringValidator class validates that a string length is within a specified range. The following example ensures that a string is between 6 and 12 characters long:

```
<?xml version="1.0"?>
<!-- validators\StringExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form id="membershipForm">
        <mx:FormItem id="fullNameItem" label="Full Name">
        <!-- Not validated -->
            <mx:TextInput id="fullNameInput"/>
        </mx:FormItem>
        <mx:FormItem id="userNameItem" label="Username">
            <mx:TextInput id="userNameInput"/>
        </mx:FormItem>
    </mx:Form>

    <mx:StringValidator source="{userNameInput}" property="text"
        minLength="6" maxLength="12"/>
</mx:Application>
```

Using the ZipCodeValidator class

The ZipCodeValidator class validates that a string has the correct length for a five-digit ZIP code, a five-digit+four-digit United States ZIP code, or a Canadian postal code.

The following example validates either a United States ZIP code or a Canadian postal code:

```
<?xml version="1.0"?>
<!-- validators\ZCExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Form id="addressForm">
        <mx:FormItem id="zipCodeItem" label="Zip Code">
            <mx:TextInput id="zipInput"/>
        </mx:FormItem>
        <mx:FormItem id="submitArea">
            <mx:Button label="Submit"/>
        </mx:FormItem>
    </mx:Form>

    <mx:ZipCodeValidator id="zipV"
        source="{zipInput}" property="text"
        domain="US or Canada"/>
</mx:Application>
```

CHAPTER 41

Formatting Data

41

This topic describes how to use data formatters, user-configurable objects that format raw data into a customized string. You often use formatters with data binding to create a meaningful display of the raw data bound to a component. This can save you time by automating data formatting tasks and by letting you easily change the formatting of fields within your applications.

Contents

Using formatters

Adobe Flex formatters are components that you use to format data into strings. Formatters perform a one-way conversion of raw data to a formatted string. You typically trigger a formatter just before displaying data in a text field. Flex includes standard formatters that let you format currency, dates, numbers, phone numbers, and ZIP codes.

All Flex formatters are subclasses of the mx.formatters.Formatter class. The Formatter class declares a `format()` method that takes a value and returns a String value.

For most formatters, when an error occurs, an empty string is returned and a string describing the error is written to the formatter's `error` property. The `error` property is inherited from the Formatter class.

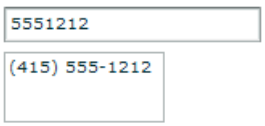
The following steps describe the general process for using a formatter:

1. Declare a formatter in your MXML code, specifying the appropriate formatting properties.
2. Call the formatter's `format()` method within the curly braces ({ }) syntax for binding data, and specify the value to be formatted as a parameter to the `format()` method.

The following example formats a phone number that a user inputs in an application using the TextInput control:

```
<?xml version="1.0"?>
<!-- formatters\Formatter2TextFields.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Declare a PhoneFormatter and define formatting parameters.-->
    <mx:PhoneFormatter id="phoneDisplay"
        areaCode="415"
        formatString="###-####" />

    <!-- Declare the input control.-->
    <mx:Label text="Enter 7 digit phone number (########):"/>
    <mx:TextInput id="myTI"/>

    <!-- Declare the control to display the formatted data.-->
    <mx:TextArea text="{phoneDisplay.format(myTI.text)}"/>
</mx:Application>
```

In this example, you use the Flex PhoneFormatter class to display the formatted phone number in a TextArea control. The following example shows the output of this application:



You do not have to use a `format()` method within curly braces ({ }) of a binding data expression; you can call this method from anywhere in your application, typically in response to an event. The following example declares a DateFormatter with an MM/DD/YYYY date format. The application then writes a formatted date to the `text` property of a TextInput control in response to the `click` event of a Button control:

```
<?xml version="1.0"?>
<!-- formatters\FormatterDateField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Declare a DateFormatter and define formatting parameters.-->
    <mx:DateFormatter id="dateFormatter"
        formatString="month: MM, day: DD, year: YYYY"/>

    <mx:Label text="Enter date (mm/dd/yyyy):"/>
    <mx:TextInput id="dob" text=""/>

    <mx:Label text="Formatted date: "/>
    <mx:TextInput id="formattedDate"
        text=""
        editable="false"/>

    <!-- Format and update the date.-->
    <mx:Button label="Format Input"
        click="formattedDate.text=dateFormatter.format(dob.text);"/>
</mx:Application>
```

Writing an error handler function

The protocol for formatters when detecting an error is to return an empty string, and to write a string describing the error condition to the `error` property of the formatter. You can check for an empty string in the return value of the `format()` method, and if found, access the `error` property to determine the cause of the error.

Alternatively, you can write an error handler function that returns an error message for a formatting error. The following example shows a simple error handler function:

```
<?xml version="1.0"?>
<!-- formatters\FormatterSimpleErrorForDevApps.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            private function formatWithError(value:Object):String {
                var formatted:String = myFormatter.format(value);
                if (formatted == "") {
                    if (myFormatter.error != null ) {
                        if (myFormatter.error == "Invalid value") {
                            formatted = ": The value is not valid.";
                        }
                        else {
                            formatted = ": The formatString is not valid.";
                        }
                    }
                }
                return formatted;
            }
        ]]>
    </mx:Script>

    <!-- Declare a formatter and specify formatting properties.-->
    <mx:DateFormatter id="myFormatter"
        formatString="MXXXMXMXMXMXM"/>

    <!-- Trigger the formatter while populating a string with data.-->
    <mx:TextInput id="myTI"
        width="75%"
        text="Your order shipped on {formatWithError('May 23, 2005')}"/>
</mx:Application>
```

In this example, you define a DateFormatter with an invalid format string. You then use the `formatWithError()` method to invoke the formatter in the TextInput control, rather than calling the Date formatter's `format()` method directly. In this example, if either the input string or the format string is invalid, the `formatWithError()` method returns an error message instead of a formatted String value.

Using the standard formatters

The following sections describe the standard formatters included with Flex.

Using the CurrencyFormatter class

The CurrencyFormatter class provides the same features as the NumberFormatter class, plus a currency symbol. It has two additional properties: `currencySymbol` and `alignSymbol`. (For more information about the NumberFormatter class, see "Using the NumberFormatter class" on page 1336.)

The CurrencyFormatter class provides basic formatting options for numeric data, including decimal formatting, thousands separator formatting, and negative sign formatting. The `format()` method accepts a Number value or a number formatted as a String value and formats the resulting string.

When a number formatted as a String is passed to the `format()` method, the function parses the string from left to right and attempts to extract the first sequence of numbers it encounters. The function parses the thousands separators and decimal separators along with their trailing numbers. The parser searches for a comma (,) for the thousands separator unless you set a different character in the `thousandsSeparatorFrom` property. The parser searches for a period (.) for the decimal separator unless you define a different character in the `decimalSeparator` property.

NOTE

When a number is provided to the `format()` method as a String value, a negative sign is recognized if it is a dash (-) immediately preceding the first number in the sequence. A dash, space, and then a first number are not interpreted as a negative sign.

Example: Using the CurrencyFormatter class

The following example uses the CurrencyFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainCurrencyFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define variable to hold the price.
            [Bindable]
            private var todaysPrice:Number=4025;
        ]]>
    </mx:Script>

    <!-- Declare a CurrencyFormatter and define parameters.-->
    <mx:CurrencyFormatter id="Price" precision="2"
        rounding="none"
        decimalSeparatorTo="."
        thousandsSeparatorTo=","
        useThousandsSeparator="true"
        useNegativeSign="true"
        currencySymbol="$"
        alignSymbol="left"/>

    <!-- Trigger the formatter while populating a string with data.-->
    <mx:TextInput text="Today's price is {Price.format(todaysPrice)}."/>
</mx:Application>
```

At run time, the following text is displayed:

Today's price is $4,025.00.

Error handling: CurrencyFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the `error` property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String.
`Invalid format`	One or more of the parameters contain an unusable setting.

Using the DateFormatter class

The DateFormatter class gives you a wide range of combinations for displaying date and time information. The `format()` method accepts a Date object, which it converts to a String based on a user-defined pattern. The `format()` method can also accept a String-formatted date, which it attempts to parse into a valid Date object prior to formatting.

The DateFormatter class has a `parseDateString()` method that accepts a date formatted as a String. The `parseDateString()` method examines sections of numbers and letters in the String to build a Date object. The parser is capable of interpreting long or abbreviated (three-character) month names, time, am and pm, and various representations of the date. If the `parseDateString()` method is unable to parse the String into a Date object, it returns `null`.

The following examples show some of the ways strings can be parsed:

```
"12/31/98" or "12-31-98" or "1998-12-31" or "12/31/1998"
"Friday, December 26, 2005 8:35 am"
"Jan. 23, 1989 11:32:25"
```

The DateFormatter class parses strings from left to right. A date value should appear before time and must be included. The time value is optional. A time signature of 0:0:0 is the Date object's default for dates that are defined without a time. Time zone offsets are not parsed.

Using Pattern strings

You provide the DateFormatter class with a string of pattern letters, which it parses to determine the appropriate formatting. You must understand how to compose the string of pattern letters to control the formatting options and the format of the string that is returned.

You compose a pattern string using specific uppercase letters: for example, YYYY/MM. The DateFormatter pattern string can contain other text in addition to pattern letters. To form a valid pattern string, you need only one pattern letter.

When you create a pattern string, you usually repeat pattern letters. The number of repeated letters determines the presentation. For numeric values, the number of pattern letters is the minimum number of digits; shorter numbers are zero-padded to this amount. For example, the four-letter pattern string "YYYY" corresponds to a four-digit year value.

In cases where there is a corresponding mapping of a text description—for instance, the name of a month—the full form is used if the number of pattern letters is four or more; otherwise, a short or abbreviated form is used, if available. For example, if you specify the pattern string "MMMM" for month, the formatter requires the full month name, such as "December", instead of the abbreviated month name, such as "Dec".

For time values, a single pattern letter is interpreted as one or two digits. Two pattern letters are interpreted as two digits.

The following table describes each of the available pattern letters:

Pattern letter	Description
Y	Year. If the number of pattern letters is two, the year is truncated to two digits; otherwise, it appears as four digits. The year can be zero-padded, as the third example shows in the following set of examples: Examples: YY = 05 YYYY = 2005 YYYYY = 02005
M	Month in year. The format depends on the following criteria: • If the number of pattern letters is one, the format is interpreted as numeric in one or two digits. • If the number of pattern letters is two, the format is interpreted as numeric in two digits. • If the number of pattern letters is three, the format is interpreted as short text. • If the number of pattern letters is four, the format is interpreted as full text. Examples: M = 7 MM= 07 MMM=Jul MMMM= July
D	Day in month. While a single-letter pattern string for day is valid, you typically use a two-letter pattern string. Examples: D=4 DD=04 DD=10
E	Day in week. The format depends on the following criteria: • If the number of pattern letters is one, the format is interpreted as numeric in one or two digits. • If the number of pattern letters is two, the format is interpreted as numeric in two digits. • If the number of pattern letters is three, the format is interpreted as short text. • If the number of pattern letters is four, the format is interpreted as full text. Examples: E = 1 EE = 01 EEE = Mon EEEE = Monday
A	am/pm indicator.

Pattern letter	Description
J	Hour in day (0-23).
H	Hour in day (1-24).
K	Hour in am/pm (0-11).
L	Hour in am/pm (1-12).
N	Minute in hour. Examples: N = 3 NN = 03
S	Second in minute. Examples: SS = 30
Other text	You can add other text into the pattern string to further format the string. You can use punctuation, numbers, and all lowercase letters. You should avoid uppercase letters because they may be interpreted as pattern letters. Example: EEEE, MMM. D, YYYY at H:NN A = Tuesday, Sept. 8, 2005 at 1:26 PM

The following table shows sample pattern strings and the resulting presentation:

Pattern	Result
`YYYY.MM.DD at HH:NN:SS`	2005.07.04 at 12:08:56
`EEE, MMM D, 'YY`	Wed, Jul 4, '05
`H:NN A`	12:08 PM
`HH o'clock A`	12 o'clock PM
`K:NN A`	0:08 PM
`YYYYY.MMMM.DD. JJ:NN A`	02005.July.04. 12:08 PM
`EEE, D MMM YYYY HH:NN:SS`	Wed, 4 Jul 2005 12:08:56

Example: Using the DateFormatter class

The following example uses the DateFormatter class in an MXML file for format a Date object as a String:

```
<?xml version="1.0"?>
<!-- formatters\MainDateFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define variable to hold the date.
            [Bindable]
            private var today:Date = new Date();
        ]]>
    </mx:Script>

    <!-- Declare a DateFormatter and define parameters.-->
    <mx:DateFormatter id="DateDisplay"
        formatString="MMMM D, YYYY"/>

    <!-- Display the date in a TextArea control.-->
    <mx:TextArea id="myTA" text="{DateDisplay.format(today)}"/>
</mx:Application>
```

At run time, the TextArea control displays the current date.

Error handling: DateFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the `error` property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	A value that is not a Date object is passed to the `format()` method. (An empty argument is allowed.)
`Invalid format`	• The `formatString` property is set to empty (""). • There is less than one pattern letter in the `formatString` property.

Using the NumberFormatter class

The NumberFormatter class provides basic formatting options for numeric data, including decimal formatting, thousands separator formatting, and negative sign formatting. The `format()` method accepts a number or a number formatted as a String value, and formats the resulting string.

When a number formatted as a String value is passed to the `format()` method, the function parses the string from left to right and attempts to extract the first sequence of numbers it encounters. The function parses the thousands separators and decimal separators along with their trailing numbers. The parser searches for a comma (,) for the thousands separator unless you set a different character in the `thousandsSeparatorFrom` property. The parser searches for a period (.) for the decimal separator unless you define a different character in the `decimalSeparator` property.

NOTE	The `format()` method recognizes a dash (-) immediately preceding the first number in the sequence as a negative number. A dash, space, and then number sequence are not interpreted as a negative number.

The `rounding` and `precision` properties of the NumberFormatter class affect the formatting of the decimal in a number. If you use both `rounding` and `precision` properties, rounding is applied first, and then the decimal length is set using the specified precision value. This lets you round a number and still have a trailing decimal; for example, 303.99 = 304.00.

Example: Using the NumberFormatter class

The following example uses the NumberFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainNumberFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define variable to hold the number.
            [Bindable]
            private var bigNumber:Number = 6000000000.65;
        ]]>
    </mx:Script>

    <!-- Declare and define parameters for the NumberFormatter.-->
    <mx:NumberFormatter id="PrepForDisplay"
        precision="0"
        rounding="up"
        decimalSeparatorTo="."
        thousandsSeparatorTo=","
        useThousandsSeparator="true"
        useNegativeSign="true"/>

    <!-- Trigger the formatter while populating a string with data.-->
    <mx:TextInput text="{PrepForDisplay.format(bigNumber)}"/>
</mx:Application>
```

At run time, the following text appears:

6,000,000,001

Error handling: NumberFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the `error` property. An error refers to a problem with the value being submitted or the format string that contains the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String.
`Invalid format`	One or more of the parameters contain an unusable setting.

Using the PhoneFormatter class

The PhoneFormatter class lets you format a phone number by adjusting the format of the area code and the subscriber code. You can also adjust the country code and configuration for international formats. The value passed into the `PhoneFormatter.format()` method must be a Number object or a String object with only digits.

The PhoneFormatter `formatString` property accepts a formatted string as a definition of the format pattern. The following table shows common options for `formatString` values. The `format()` method for the PhoneFormatter accepts a sequence of numbers. The numbers correspond to the number of placeholder (#) symbols in the `formatString` value. The number of placeholder symbols in the `formatString` property and the number of digits in the `format()` method value must match.

formatString value	Input	Output
`###-####`	1234567	(xxx) 456-7890
`(###) ### ####`	1234567890	(123) 456-7890
`###-###-####`	11234567890	123-456-7890
`#(###) ### ####`	11234567890	1(123) 456 7890
`#-###-###-####`	11234567890	1-123-456-7890
`+###-###-###-####`	1231234567890	+123-123-456-7890

In the preceding table, dashes (-) are used as separator elements where applicable. You can substitute period (.) characters or blank spaces for the dashes. You can change the default allowable character set as needed using the `validPatternChars` property. You can change the default character that represents a numeric placeholder by using the `numberSymbol` property (for example, to change from # to $).

NOTE	A shortcut is provided for the United States seven-digit format. If the `areaCode` property contains a value and you use the seven-digit format string, a seven-digit format entry automatically adds the area code to the string returned. The default format for the area code is (###). You can change this using the `areaCodeFormat` property. You can format the area code any way you want as long as it contains three number placeholders.

Example: Using the PhoneFormatter class

The following example uses the PhoneFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainPhoneFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define variable to hold the phone number.
            [Bindable]
            private var newNumber:Number = 1234567;
        ]]>
    </mx:Script>

    <!-- Declare a PhoneFormatter and define formatting parameters.-->
    <mx:PhoneFormatter id="PhoneDisplay"
        areaCode="415"
        formatString="###-####"/>

    <!-- Trigger the formatter while populating a string with data-->
    <mx:TextInput id="myTI"
        initialize="myTI.text=PhoneDisplay.format(newNumber);"/>
</mx:Application>
```

At run time, the following text is displayed:

(415) 123-4567

Error handling: PhoneFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the `error` property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	• An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String. • The value contains a different number of digits than what is specified in the format string.
`Invalid format`	• One or more of the characters in the `formatString` do not match the allowed characters specified in the `validPatternChars` property. • The `areaCodeFormat` property is specified but does not contain exactly three numeric placeholders.

Using the ZipCodeFormatter class

The ZipCodeFormatter class lets you format five-digit or nine-digit United States ZIP codes and six-character Canadian postal codes. The ZipCodeFormatter class's `formatString` property accepts a formatted string as a definition of the format pattern. The `formatString` property is optional. If it is omitted, the default value of `#####` is used.

The number of digits in the value to be formatted and the value of the `formatString` property must be five or nine for United States ZIP codes, and six for Canadian postal codes.

The following table shows common `formatString` values, input values, and output values:

formatString value	Input	Output	Format
`#####`	94117, 941171234	94117, 94117	Five-digit U.S. ZIP code
`#####-####`	941171234, 94117	94117-1234, 94117-0000	Nine-digit U.S. ZIP code
`### ###`	A1B2C3	A1B 2C3	Six-character Canadian postal code

For United States ZIP codes, if a nine-digit format is requested and a five-digit value is supplied, `-0000` is appended to the value to make it compliant with the nine-digit format. Inversely, if a nine-digit value is supplied for a five-digit format, the number is truncated to five digits.

For Canadian postal codes, only a six-digit value is allowed for either the `formatString` or the input value.

NOTE

For United States ZIP codes, only numeric characters are valid. For Canadian postal codes, alphanumeric characters are allowed. Alphabetic characters must be in uppercase.

Example: Using the ZipCodeFormatter class

The following example uses the ZipCodeFormatter class in an MXML file:

```
<?xml version="1.0"?>
<!-- formatters\MainZipFormatter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define variable to hold the ZIP code.
            [Bindable]
            private var storedZipCode:Number=123456789;
        ]]>
    </mx:Script>

    <!-- Declare a ZipCodeFormatter and define parameters.-->
    <mx:ZipCodeFormatter id="ZipCodeDisplay"
        formatString="#####-####"/>

    <!-- Trigger the formatter while populating a string with data.-->
    <mx:TextInput text="{ZipCodeDisplay.format(storedZipCode)}"/>
</mx:Application>
```

At run time, the following text is displayed:

12345-6789

Error handling: ZipCodeFormatter class

If an error occurs, Flex returns an empty string and saves a description of the error in the `error` property. An error refers to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

Value of error property	When error occurs
`Invalid value`	• An invalid numeric value is passed to the `format()` method. The value should be a valid number in the form of a Number or a String, except for Canadian postal codes, which allow alphanumeric values. • The number of digits does not match the allowed digits from the `formatString` property.
`Invalid format`	• One or more of the characters in the `formatString` do not match the allowed characters specified in the `validFormatChars` property. • The number of numeric placeholders does not equal 9, 5, or 6.

PART 6

Data Access and Interconnectivity

6

This part describes the Adobe Flex 2 features that let you work with external data.

The following topics are included:

CHAPTER 42

Accessing Server-Side Data 42

This topic introduces the Adobe Flex 2 features that let you work with external data. It also specifies whether features are available in Adobe Flex 2 SDK or Adobe Flex Data Services. Flex Data Services runs on standard J2EE platforms and servlet containers.

Contents

About Flex data access

Adobe Flex 2 provides a new messaging infrastructure for building data-rich applications. This infrastructure underlies the three Flex features that are designed for moving data to and from applications: RPC services, the Data Management Service, and the Message Service.

The following table describes the Flex data access features:

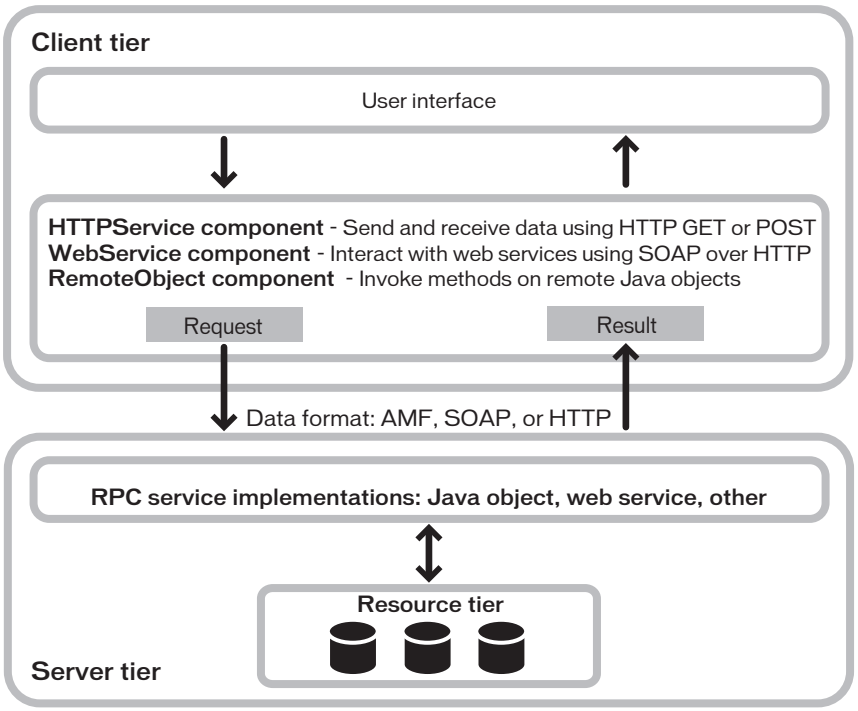
Feature	Description	Availability
RPC services	Provides a call and response model for accessing external data. This feature lets you create applications that make asynchronous requests to remote services that process the requests and then return data to your Flex application. With Flex 2 SDK only, you can call services directly, but you cannot use server-side features that require Flex Data Services. For more information, see "About RPC services" on page 1346.	Flex 2 SDK Flex Data Services
Data Management Service	Provides data synchronization between application tiers, real-time data updates, data replication, occasionally connected application services, and integration with data sources through adapters. This feature lets you create applications that work with distributed data, and lets you manage large collections of data and nested data relationships, such as one-to-one and one-to-many relationships. For more information, see "About the Data Management Service" on page 1348.	Flex Data Services
Message Service	Provides messaging services for collaborative and real-time applications. This feature lets you create applications that can send messages to and receive messages from other applications, including Flex applications and Java Message Service (JMS) applications. For more information, see "About messaging" on page 1349.	Flex Data Services

About RPC services

The RPC services feature is designed for applications in which a call and response model is a good choice for accessing external data. This feature lets you make asynchronous requests to remote services that process the requests and then return data directly to your Flex application.

A client-side RPC component, which you can create in MXML or ActionScript, calls a remote service, and then stores response data from the service in an ActionScript object from which you can easily obtain the data. The implementation of an RPC service can be an HTTP URL, which uses HTTP POST or GET, a SOAP-compliant web service, or a Java object in a Java web application. The client-side RPC components are the HTTPService, WebService, and RemoteObject components.

The following diagram provides a simplified view of how RPC components interact with RPC services:



Using RPC components with Flex 2 SDK only

You can use Flex 2 SDK without Flex Data Services to create nonenterprise applications that call HTTP services or web services directly, without going through a server-side proxy service. You cannot use RemoteObject components without Flex Data Services.

By default, Flash Player blocks access to any host that is not exactly equal to the one used to load an application. Therefore, an RPC service must either be on the server hosting your application, or the remote server that hosts the RPC service must define a crossdomain.xml file.

A *crossdomain.xml file* is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from certain domains, or from all domains. The crossdomain.xml file must be in the web root of the server that the Flex application is contacting.

For more information, see Chapter 44, "Understanding RPC Components," on page 1399.

Using RPC components with Flex Data Services

Use Flex Data Services with RPC components when you want to provide enterprise functionality, such as proxying of service traffic from different domains, client authentication, whitelists of permitted RPC service URLs, server-side logging, localization support, and centralized management of RPC services. Flex Data Services also gives you the option of using RemoteObject components to access remote Java objects without configuring them as SOAP-compliant web services.

When you use Flex Data Services, instead of contacting services directly, RPC components contact destinations. *Destinations* are manageable service endpoints that you manage through a server-side XML-based configuration file.

For more information, see Chapter 44, "Understanding RPC Components," on page 1399.

About the Data Management Service

The Data Management Service is a Flex Data Services feature that lets you create applications that work with distributed data. This feature lets you build applications that provide data synchronization, data replication, and occasionally connected application services. Additionally, you can manage large collections of data and nested data relationships, such as one-to-one and one-to-many relationships, and use data adapters to integrate with data resources.

A client-side DataService component, which you can create in MXML or ActionScript, calls methods on a server-side Data Management Service destination to perform such activities as filling client-side data collections with data from remote data sources and synchronizing the client and server versions of data.

About messaging

Messaging is a Flex Data Services feature that lets you create collaborative and real-time messaging applications. It provides a client-side API for passing text and objects to and from messaging clients, and server-side configuration of message destinations.

You can create messaging components in MXML or ActionScript that connect to server-side message destinations, send messages to those destinations, and receive messages from other messaging clients. Components that send messages are called producers, while those that receive messages are called consumers. Messages are sent over transport *channels*, which use specific transport protocols, such as the Realtime Message Protocol (RTMP) and Action Message Format (AMF).

Messaging clients can be Flex applications or other types of applications, such as Java Message Service (JMS) applications. JMS is a Java API that lets applications create, send, receive, and read messages. JMS applications can publish and subscribe to the same message destinations as Flex applications. However, you can create a wide variety of messaging applications using just Flex messaging.

Message destinations are managed through a server-side XML-based configuration file. You can configure client authentication for destinations. You can also configure server-side logging for messaging traffic.

Using Flex Data Services with Flex Builder

Flex Builder offers two ways to work with Flex Data Services. You can either compile an application in Flex Builder or you can save your uncompiled application (MXML file) to a Flex Data Services web application on the server and compile the application when the page is viewed. When you compile a Flex Data Services project in Flex Builder, Flex Builder automatically saves the compiled application (SWF file) to a Flex Data Services web application on the server whenever you compile the application.

The two options for configuring a Flex Data Services project in Flex Builder are available on the Create a Flex Project dialog when you select New > Flex Project from the Flex Builder file menu.

When you create a Flex Data Services project in Flex Builder, Flex Builder either creates a new directory with same name as your project or uses an existing directory that has that name. That directory is a subdirectory of the root folder that you specify. A default application file (MXML file) of the same name is also created. If desired, you can set a different MXML as the default application file by selecting the file in the Flex Builder Navigator tree and right clicking the Set as Default Application option.

For applications that compile locally, Flex Builder saves MXML files the local Flex Builder project directory and saves SWF files and HTML wrapper files on the server. Creating a project that compiles locally is a good idea if you plan to deploy your finished application as a SWF file and HTML wrapper file rather than an MXML file.

For applications the compile on the server, MXML files are saved on the server. No HTML wrapper file is generated and saved when the application is compiled.

When using a Flex Builder project for a Flex Data Services application, you manage resources such as server-side Java classes just as you do when not using Flex Builder. You can specify the location of additional client-side source files, such as ActionScript classes, in the source path panel of the Flex Builder project properties dialog; by default, Flex Builder includes the WEB-INF\flex\user_classes directory for applications the compile on the server.

NOTE

Regardless of which option you choose for a Flex Data Services project in Flex Builder, you must specify a valid Flex Data Services root folder and root URL. These values should map the root of a Flex Data Services web application. Before compiling your Flex application in Flex Builder, the corresponding Flex Data Services web application must be started.

For more information about Flex Builder projects, see Chapter 3, "Working with Projects" in *Using Flex Builder 2*.

CHAPTER 43

Configuring Data Services

43

The services in Adobe Flex Data Services use a common set of configuration features in addition to features that are service-specific. This topic describes the following features that are common to all of the service types:

- Service configuration files
- Message channels
- Data serialization
- Authentication and authorization
- Server-side logging
- Session data
- Software clustering
- Run-time service monitoring and management
- Custom error handling
- Class loading

Contents

About service configuration files

You configure the Remoting Service, Proxy Service, Message Service, and Data Management Service in the `<services>` section of the Data Services configuration file, services-config.xml. The default location of this file is the WEB-INF/flex directory of your Flex Data Services web application. This location is set in the configuration for the MessageBrokerServlet in the WEB-INF/web.xml file.

As a convenience, in the services-config.xml file, you can include files that contain service definitions by reference. Adobe generally includes the Remoting Service, Proxy Service, Message Service, and Data Management Service by reference.

The following table describes the typical way that Adobe sets up configuration files. Commented versions of these files are available in the resources/config subdirectory of the Flex Data Services installation directory; you can use those files as templates for your own configurations.

Filename	Description
services-config.xml	The top-level Flex Data Services configuration file. This file usually contains security constraint definitions, channel definitions, and logging settings that each of the services can use. It can contain service definitions inline or include them by reference, which Adobe typically does. Adobe generally defines the services in the remoting-config.xml, proxy-config.xml, messaging-config.xml, and data-management-config.xml files.
remoting-config.xml	The Remoting Service configuration file, which defines Remoting Service destinations for working with remote objects. For information about configuring the Remoting Service, see Chapter 46, "Configuring RPC Services," on page 1451.
proxy-config.xml	The Proxy Service configuration file, which defines Proxy Service destinations for working with web services and HTTP services (REST services). For information about configuring the Proxy Service, see Chapter 46, "Configuring RPC Services," on page 1451.

Filename	Description
messaging-config.xml	The Message Service configuration file, which defines Message Service destinations for performing publish subscribe messaging. For information about configuring the Messaging Service, see Chapter 49, "Configuring the Message Service," on page 1483.
data-management-config.xml	The Data Management Service configuration file, which defines Data Management Service destinations for working with Data Management Service destinations. For information about configuring the Data Management Service, see Chapter 52, "Configuring the Data Management Service," on page 1517.

When you include a file by reference, the content of the referenced file must conform to the appropriate XML structure for the service. the file-path value is relative to the location of the services-config.xml file. The following example shows service definitions included by reference:

```
<services>
  <!-- REMOTING SERVICE -->
  <service-include file-path="remoting-config.xml"/>

  <!-- PROXY SERVICE -->
  <service-include file-path="proxy-config.xml"/>

  <!-- MESSAGE SERVICE -->
  <service-include file-path="messaging-config.xml"/>

  <!-- DATA MANAGEMENT SERVICE -->
  <service-include file-path="data-management-config.xml"/>
</services>
```

Data Service configuration file syntax

The following table describes the XML elements of the services-config.xml file. The root element is the `services-config` element.

XML element	Description
`services`	Contains definitions of individual data services or references to other XML files that that contain service definitions. Adobe includes a separate configuration file for each type of standard service; these include the Proxy Service, Remoting Service, Message Service, and Data Management Service. The `services` element is declared at the top level of the configuration as a child of the root element, `services-config`. For information about configuring specific types of services, see the following topics: • Chapter 46, "Configuring RPC Services," on page 1451 • Chapter 49, "Configuring the Message Service," on page 1483 • Chapter 52, "Configuring the Data Management Service," on page 1517
`services/service-include`	Specifies the full path to an XML file that contains the configuration elements for a service definition. **Attributes:** `file-path` Path to the XML file that contains a service definition.
`services/service`	Contains the definition for a data service, such as the Proxy Service, Remoting Service, Message Service, Data Management Service, or a custom service. (Optional) You can use the `service-include` element to include a file that contains a service definition by reference instead of inline in the services-config.xml file.
`services/service/properties`	Contains elements for service-level properties.
`services/service/adapters`	Contains definitions for service adapters that are referenced in a destination to provide specific types of functionality.

XML element	Description
`services/service/adapters/adapter-definition`	Contains a service adapter definition. Each type of service has its own set of adapters that are relevant to that type of service. For example, Data Management Service destinations can use the Java adapter or the ActionScript object adapter. **Attributes:** • `id` Identifier of an adapter, which you use to reference the adapter inside a destination definition. • `class` Fully qualified name of the Java class that provides the adapter functionality. • `default` Boolean value that indicates whether this adapter is the default adapter for service destinations. The default adapter is used when you do not explicitly reference an adapter in a destination definition.
`services/service/default-channels`	Contains references to a service's default channels. The default channels are used when a channel is not explicitly referenced in a destination definition. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried.
`services/service/default-channels/channel`	Contains references to the `id` of a channel definition. **Attributes:** `ref` The `id` value of a channel definition.
`services/service/destination`	Contains a destination definition.
`services/service/destination/adapter`	Contains a reference to a service adapter. If this element is omitted, the destination uses the default adapter.
`services/service/destination/properties`	Contains destination properties. The properties available depend on the type of service, which is determined by the service class specified.

XML element	Description
`services/service/destination/channels`	Contains references to the channels that the service can use for data transport. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried.
`services/service/destination/channels/channel`	Contains references to the `id` value of a channel. Channels are defined in the `channels` element at the top level of the configuration as a child of the root element, `services-config`.
`services/service/destination/security`	Contains references to security constraint definitions and login command definitions that are used for authentication and authorization. This element can also contain complete security constraint definitions instead of references to security constraints that are defined globally in the top-level security element. For more information, see "Securing destinations" on page 1379.
`services/service/destination/security/security-constraint`	Contains references to the `id` value of a security constraint definition or contains a security constraint definition. **Attributes:** `ref` The `id` value of a `security-constraint` element defined in the `security` element at the top-level of the services configuration. `id` Identifier of a security constraint when you define the actual security constraint in this element.
`services/service/destination/security/security-constraint/auth-method`	Specifies the authentication method that the security constraint employs. A security constraint can use either basic or custom authentication. The valid values are `basic` and `custom`.
`services/service/destination/security/security-constraint/roles`	Specifies the names of roles that are defined in the application server's user store.

XML element	Description
`services/service/destination/security/login-command`	Contains references to the `id` value of a login command definition that is used for performing custom authentication. **Attributes:** `ref` The `id` value of a login command definition. For more information, see "Custom authentication" on page 1383
`security`	Contains security constraint definitions and login command definitions for authentication and authorization. For more information, see "Securing destinations" on page 1379.
`security/security-constraint`	Defines a security constraint.
`security/security-constraint/auth-method`	Specifies the authentication method that the security constraint employs. A security constraint can use basic or custom authentication.
`security/security-constraint/roles`	Specifies the names of roles that are defined in the application server's user store.
`security/login-command`	Defines a login command that is used for custom authentication. **Attributes:** • `class` Fully qualified classname of a login command class. • `server` Application server on which custom authentication is performed. For more information, see "Custom authentication" on page 1383.
`channels`	Contains the definitions of message channels that are used to transport data between the server and clients. For more information, see "Configuring message channels" on page 1360.
`channels/channel-definition`	Defines a message channel that can be used to transport data. **Attributes:** • `id` Identifier of the channel definition. • `class` Fully qualified classname of a channel class.

XML element	Description
`channels/channel-definition/endpoint`	Specifies the endpoint URI and the endpoint class of the channel definition. **Attributes:** • `uri` Endpoint URI. • `class` Fully qualified name of the endpoint class.
`channels/channel-definition/properties`	Contains the properties of a channel definition. The properties available depend on the type of channel specified. For more information, see "Configuring message channels" on page 1360.
`clusters`	Contains cluster definitions, which configure software clustering across multiple hosts. For more information, see "Using software clustering" on page 1388.
`clusters/cluster`	Defines a cluster. **Attributes:** `id` Identifier of the cluster definition. `properties` The name of a JGroups properties file.
`logging`	Contains server-side logging configuration. For more information, see "Configuring server-side service logging" on page 1384.
`logging/target`	Specifies the logging target class and the logging level. **Attributes:** • `class` Fully qualified logging target classname. • `level` The logging level; see "Configuring server-side service logging" on page 1384.
`logging/target/properties`	Contains the logging properties listed in the following rows. All of these properties are optional.
`logging/target/properties/prefix`	Specifies a prefix to include on log messages.
`logging/target/properties/includeDate`	Boolean value that indicates whether to include the date in log messages.
`logging/target/properties/includeTime`	Boolean value that indicates whether to include the time in log messages.

XML element	Description
`logging/target/properties/includeLevel`	Boolean value that indicates whether to include the log level in log messages.
`logging/target/properties/includeCategory`	Boolean value that indicates whether to include the log category in log messages. The log categories correspond to the values of the `filters`/`pattern` elements described below. Notice that the server logs against specific logging categories, not wildcard patterns.
`logging/target/filters`	Contains patterns to match against when creating log messages. Only messages that match specified patterns are logged.
`logging/target/filters/pattern`	A pattern to match against when creating log messages. Only messages that match specified patterns are logged. The valid pattern values are listed in "Configuring server-side service logging" on page 1384.
`system`	System-wide settings that do not fall into a previous category. In addition to locale information, it also contains redeployment and watch file settings.
`system/locale`	(Optional) Locale string; for example, `"en"`, `"de"`, `"fr"`, and `"es"` are valid locale strings.
`system/default-locale`	The default locale string. If no `default-locale` element is provided, a base set of English error messages is used.
`system/redeploy`	Support for web application redeployment when configuration files are updated. This feature works with J2EE application server web application redeployment. The `touch-file` value is the file used by your application server to force web redeployment. Check the application server to confirm what the `touch-file` value should be.
`system/redeploy/enabled`	Boolean value that indicates whether redeployment is enabled.
`system/redeploy/watch-interval`	Number of seconds to wait before checking for changes to configuration files.

XML element	Description
`system/redeploy/watch-file`	A Data Services configuration file to be watched for changes. The `watch-file` value must start with `{context.root}` or be an absolute path. The following example uses `{context.root}`: `{context.root}/WEB-INF/flex/data-management-config.xml`
`system/redeploy/touch-file`	The file that the application server uses to force redeployment of a web application. The `touch-file` value must start with `{context.root}` or be an absolute path. Check the application server documentation to determine what the `touch-file` value should be. For JRun, the `touch-file` value is the web.xml file, as the following example shows: `{context.root}/WEB-INF/web.xml`

Configuring message channels

Flex Data Services transports messages to and from service destinations over message channels that are part of the Flex messaging system. You can pair any type of service with any type of channel.

Flex Data Services includes support for AMF, HTTP, and RTMP channels and for secure AMF, HTTP, and RTMP channels. If you require data security, you should use one of the secure channels. The secure AMF and HTTP channels transport data by using HTTP over Secure Socket Layer (HTTPS). The secure RTMP channel transports data by using RTMP over Transport Layer Security (TLS).

You create channel definitions in the Data Services configuration file. The endpoint URI of each channel definition must be unique.

You can configure an AMF polling channel to repeatedly poll the AMF endpoint to create client-pull message consumers. The interval at which the polling occurs is configurable on the channel.

The following example shows an AMF channel definition. The channel configuration is preceded by a destination that references it.

```
...
  <destination id="MyTopic">
...
    <channels>
      <channel ref="samples-amf"/>
```

```
    </channels>
...
  </destination>
...

  <channel-definition id="samples-amf"
    type="mx.messaging.channels.AMFChannel">
    <endpoint uri="/myapp/messagebroker/amf" port="8100"
      type="flex.messaging.endpoints.AmfEndpoint"/>
  </channel-definition>
</channels>
...
```

You can also assign channels on the client at run time by creating a ChannelSet object that contains one or more Channel objects, and assigning the ChannelSet to a service component, as the following example shows:

```
...
// Create a ChannelSet.
var cs:ChannelSet = new ChannelSet();
// Create a Channel.
var customChannel:Channel = new AMFChannel("my-polling-amf", endpointUrl);
// Add the Channel to the ChannelSet.
cs.addChannel(customChannel);
// Assign the ChannelSet to a RemoteObject instance.
myRemoteObject.channelSet = cs;
...
```

When you configure channels at run time, you can dynamically control the endpointURL. You can add as many Channel objects to a ChannelSet as you want; the ChannelSet object searches through the set to find a channel to which it can connect. It is a best practice to specify an ID that the destination recognizes, but you can pass null if you do not want to do ID checking.

In addition to the searching behavior provided by ChannelSet, the Channel class defines a `failoverURIs` property. This property lets you configure a channel directly in ActionScript that causes failover across this array of endpoint URLs when it tries to connect to its destination.

The connection process involves searching to the first channel and trying to connect to it. If this fails, and the channel defines failover URIs, each is attempted before the channel gives up and the ChannelSet searches to the next available channel. If no channel in the set can connect, any pending unsent messages generate faults on the client.

Flex Data Services supports the following message channels:

Message channel	Description

RTMP	Channel that you use to connect to the RTMP endpoint, which supports real-time messaging and server-pushed broadcasts.
	The following example shows an RTMP channel definition:
	`<channel-definition id="my-rtmp"`
	`  class="mx.messaging.channels.RTMPChannel">`
	`  <endpoint uri="rtmp://{server.name}:2035"`
	`  class="flex.messaging.endpoints.RTMPEndpoint"/>`
	`  <properties>`
	`  <idle-timeout-minutes>20</idle-timeout-minutes>`
	`  </properties>`
	`</channel-definition>`
	You cannot use server.port or context.root tokens in RTMP endpoint configuration. You can use the server.name token, but not when using clustering.
	If you have a firewall in place, you must open the port that you assign to the RTMP endpoint to allow RTMP traffic. You cannot use HTTP tunnelling for RTMP traffic.
	To provide secure data transmission that uses Secure Socket Layer (SSL), use the Secure RTMP channel instead of the standard RTMP channel.
	The optional `max-worker-threads` element in the `properties` section of a RTMP channel definition lets you set a fixed RTMP endpoint worker pool size. The default behavior is to use a cached thread pool that grows as needed with no upper limit. The following example shows a `max-worker-threads` element:
	`<channel-definition id="rtmp-ac"`
	`  class="mx.messaging.channels.RTMPChannel">`
	`  <endpoint uri="rtmp://10.132.64.63:2266/eqa/rtmpac"`
	`  class="flex.messaging.endpoints.RTMPEndpoint"/>`
	`  <properties>`
	`  <max-worker-threads>10</max-worker-threads>`
	`...`

AMF	Channel that you use to send AMF-encoded messages to the AMF endpoint, which supports AMF requests and responses sent over HTTP. When using this channel for publish-subscribe messaging, you usually set the `polling` property to `true` in the channel definition. You can also configure the polling interval in the channel definition.
	The following example shows an AMF channel definition:
	`<channel-definition id="samples-amf"`
	`  type="mx.messaging.channels.AMFChannel">`
	`  <endpoint uri="http://{server.name}:8100/myapp/messagebroker/`
	`  amf" type="flex.messaging.endpoints.AmfEndpoint"/>`
	`</channel-definition>`

Message channel	Description

AMF polling	Channel that you use to send AMF-encoded messages to the AMF endpoint, which supports AMF requests and responses sent over HTTP. When using this channel for publish-subscribe messaging, you set the `polling` property to `true` in the channel definition. You can also configure the polling interval in the channel definition.
	The following example shows an AMF Polling channel definition with polling enabled:
	`<channel-definition id="samples-polling-amf"`
	`type="mx.messaging.channels.AMFChannel">`
	`<endpoint uri="http://{server.name}:8100/dev/messagebroker/`
	`amfpolling" type="flex.messaging.endpoints.AmfEndpoint"/>`
	`<properties>`
	`<polling-enabled>true</polling-enabled>`
	`<polling-interval-seconds>8</polling-interval-seconds>`
	`</properties>`
	`</channel-definition>`

Message channel	Description

Secure RTMP	Channel that you use to connect to the RTMP endpoint over using Transport Layer Security (TLS). This channel supports real-time messaging and server-pushed broadcasts. This channel requires a digital certificate, and contains child elements for specifying a keystore filename and password. The following example shows a secure RTMP channel definition. The keystore file would not necessarily be in the location shown.
	`<channel-definition id="my-rtmps"`
	`class="mx.messaging.channels.SecureRTMPChannel">`
	`<endpoint uri="rtmps://{server.name}:2099"`
	`class="flex.messaging.endpoints.SecureRTMPEndpoint"/>`
	`<properties>`
	`<idle-timeout-minutes>30</idle-timeout-minutes>`
	`<keystore-file>C:/somedir/flex/dataservices/apps/j2ee/dev/`
	`WEB-`
	`INF/flex/keys/server.keystore</keystore-file>`
	`<keystore-password>`*password*`</keystore-password>`
	`</properties>`
	`</channel-definition>`
	You can create public and private key pairs and self-signed certificates with the Java keytool utility. The user name of self-signed certificates must be set to `localhost` or the IP address on which the RTMPS endpoint is available. For information about key and certificate creation and management, see the Java keytool documentation at http://java.sun.com.
	Optionally, you can store a keystore password in a separate file, possibly on removable media. You only need to specify one keystore-password or keystore-password-file. The following example shows a `keystore-password-file` element:
	`<keystore-password-file>a:\password</keystore-password-file>`
	You can also specify an alternate JVM algorithm instead of the default JVM algorithm when using SSL. The default JVM algorithm can be controlled via the ssl.KeyManagerFactory.algorithm security property. In general, you should use the default algorithm and this property can be omitted. The following values are valid:
	• `<algorithm>Default</algorithm>` to explicitly uses the default JVM algorithm.
	• `<algorithm>SunX509</algorithm>` to use Sun's algorithm; requires the Sun security provider.
	• `<algorithm>IbmX509</algorithm>` to use IBM's algorithm; requires the IBM security provider.
	You cannot use server.port or context.root tokens in the RTMP endpoint configuration. You can use the server.name token, but not when using clustering.
	If you have a firewall in place, you must open the port that you assign to the RTMP endpoint to allow RTMP traffic. You cannot use HTTP tunnelling for RTMP traffic.

Message channel	Description

Secure AMF	Channel that is similar to the AMF channel, but uses HTTPS instead of HTTP. Uses a different endpoint and class than the AMF channel.
	The following example shows a secure AMF channel definition:
	`<channel-definition id="my-secure-amf"`
	`  class="mx.messaging.channels.SecureAMFChannel">`
	`  <endpoint uri="https://{server.name}:9100/dev/messagebroker/`
	`  amfsecure"`
	`  class="flex.messaging.endpoints.SecureAMFEndpoint"/>`
	`</channel-definition>`

HTTP	Channel that you use to connect to the HTTP endpoint, which supports HTTP requests and responses. This channel uses a text-based (XML) message format.
	The following example shows an HTTP channel definition:
	`<channel-definition id="my-http"`
	`  class="mx.messaging.channels.HTTPChannel">`
	`  <endpoint uri="http://{server.name}:8100/dev/messagebroker/`
	`  http"`
	`  class="flex.messaging.endpoints.HTTPEndpoint"/>`
	`</channel-definition>`
	Note: You should not use types that implement the IExternalizable interface (used for custom serialization) with the HTTPChannel if precise by-reference serialization is required. When you do this, references between recurring objects are lost and appear to be cloned at the endpoint. For more information about the IExternalizable interface, see "Using custom serialization" on page 1375.
	You can specify a URL in the `redirectURL` property in the `properties` section of a channel definition when using HTTP-based channel endpoints. This is a legacy feature that lets you redirect a request to another URL if the MIME type of the request wasn't correct.

Secure HTTP	Channel similar to the HTTP channel, but you use HTTPS instead of HTTP. Uses a different channel class than the HTTP channel.
	The following example shows a secure HTTP channel definition:
	`<channel-definition id="my-secure-http"`
	`  class="mx.messaging.channels.SecureHTTPChannel">`
	`  <endpoint uri=`
	`  "https://{server.name}:9100/dev/messagebroker/`
	`  httpsecure"`
	`  class="flex.messaging.endpoints.SecureHTTPEndpoint"/>`
	`</channel-definition>`

Configuring channel endpoints

You can protect HTTP- and RTMP- based channel endpoints by using whitelists and blacklists that list specific firewall, router, or web server IP addresses. A whitelist contains client IP addresses that are permitted to access endpoints. A blacklist contains client IP addresses that are restricted from accessing endpoints.

The blacklist takes precedence over the whitelist in the event that the client IP address is a member of both the whitelist and blacklist.

The `whitelist` and `blacklist` elements can contain 0-N `ip-address` and/or `ip-address-pattern` elements. The `ip-address` element supports simple IP address matching, allowing individual bytes in the address to be wildcarded by using an asterisk character (*).

The `ip-address-pattern` element supports regular expression pattern matching of IP addresses. This allows for powerful range-based IP address filtering.

The following example shows a whitelist and a blacklist:

```
<whitelist>
  <ip-address-pattern>237.*</ip-address-pattern>
  <ip-address>10.132.64.63</ip-address>
</whitelist>

<blacklist>
  <ip-address>10.60.147.*</ip-address>
  <ip-address-pattern>10\\.132\\.17\\.5[0-9]{1,2}</ip-address-pattern>
</blacklist>
```

By default, the RTMP server that an RTMP endpoint starts up binds to all local network interface cards (NICs) on the specified port (the port is based upon the endpoint URL). You can specify a `<bind-address>xxx.xxx.xxx.xxx</bind-address>` element in the RTMP channel's `properties` section to bind the RTMP server to a specific local NIC on startup. The scenario for this is a machine with an internally facing NIC and a public NIC. You can bind the RTMP server to just the internal NIC, for example, to protect it from public access.

By default, the RTMP server's accept socket uses a platform default to control the size of the queue of pending connections to accept. You can use the `accept-backlog` element in the RTMP channel's propertie section to control the size of this queue.

Serializing data

This section contains information on serialization to and from ActionScript objects on the client and Java objects on the server, as well as serialization to and from ActionScript objects on the client and SOAP and XML schema types.

Converting data from ActionScript to Java

When method parameters send data from a Flex application to a Java object, the data is automatically converted from an ActionScript data type to a Java data type. When Flex searches for a suitable method on the Java object, it uses further, more lenient conversions to find a match.

Simple data types on the client, such as Boolean and String values, typically match exactly a remote API. However, Flex attempts some simple conversions when searching for a suitable method on a Java object.

An ActionScript Array can index entries in two ways. A *strict Array* is one in which all indexes are Numbers. An *associative Array* is one in which at least one index is based on a String. It is important to know which type of Array you are sending to the server, because it changes the data type of parameters that are used to invoke a method on a Java object.

Simple data type conversions

The following table lists the supported ActionScript (AMF 3) to Java conversions for simple data types:

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
Array (dense)	java.util.List	java.util.Collection, `Object`[ ] (native array) If the type is an interface, it is mapped to the following interface implementations • List becomes ArrayList • SortedSet becomes TreeSet • Set becomes HashSet • Collection becomes ArrayList A new instance of a custom Collection implementation is bound to that type.
Array (sparse)	java.util.Map	java.util.Collection, native array
Boolean String of `"true"` or `"false"`	java.lang.Boolean	Boolean, boolean, String
flash.utils.ByteArray	byte []	
flash.utils.IExternalizable	java.io.Externalizable	

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
Date	java.util.Date (formatted for Coordinated Universal Time (UTC))	java.util.Date, java.util.Calendar, java.sql.Timestamp, java.sql.Time, java.sql.Date
int/uint	java.lang.Integer	java.lang.Byte, java.lang.Double, java.lang.Float, java.lang.Long, java.lang.Short, java.math.BigDecimal, String, primitive types of byte, double, float, long, and short
null	null	primitives
number	java.lang.Double	java.lang.Byte, java.lang.Double, java.lang.Float, java.lang.Long, java.lang.Short, java.math.BigDecimal, String, 0 (zero) if null is sent, primitive types of byte, double, float, long, and short
Object (generic)	java.util.Map	If a Map interface is specified, Flex Data Services creates a new java.util.HashMap for java.util.Map and a new java.util.TreeMap for java.util.SortedMap.
String	java.lang.String	java.lang.String, java.lang.Boolean, java.lang.Number
typed Object	typed Object when you use [RemoteClass] metadata that specifies remote classname. Bean type must have a public no args constructor.	typed Object
undefined	null	null for Object, default values for primitives

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
XML	org.w3c.dom.Document	org.w3c.dom.Document
XMLDocument (legacy XML type)	org.w3c.dom.Document	org.w3c.dom.Document You can enable legacy XML support for the XMLDocument type on any channel defined in the services-config.xml file. This setting is only important for sending data from the server back to the client; it controls how org.w3c.dom.Document instances are sent to ActionScript. For more information, see "Providing legacy AMF serialization on a channel" on page 1372.

Primitive values cannot be set to null in Java. When passing Boolean and Number values from the client to a Java object, Flex interprets null values as the default for primitive types; for example, 0 for double, float, long, int, short, byte, \u0000 for char, and false for Boolean. Only primitive Java types get default values.

Flex Data Services handles java.lang.Throwable objects like any other typed object. They are processed with rules that look for getters and setters, and typed objects are returned to the client. The rules are like normal bean rules except they look for getters for read-only properties, and getters and setters for read-write properties. This lets you get all of the information from a Java exception. If you require legacy behavior for Throwable objects, you can set the `legacy-throwable` property to `true` on a channel; for more information, see "Providing legacy AMF serialization on a channel" on page 1372.

You can pass strict Arrays as parameters to methods that expect an implementation of the java.util.Collection or java.lang.reflect.Array APIs.

A Java Collection can contain any number of Object types, whereas a Java Array requires that entries are the same type (for example, java.lang.Object[ ], and int[ ]).

Flex also converts ActionScript strict Arrays to appropriate implementations for common Collection API interfaces. For example, if an ActionScript strict Array is sent to the Java object method `public void addProducts(java.util.Set products)`, Flex converts it to a java.util.HashSet instance before passing it as a parameter, because HashSet is a suitable implementation of the java.util.Set interface. Similarly, Flex passes an instance of java.util.TreeSet to parameters typed with the java.util.SortedSet interface.

Flex passes an instance of java.util.ArrayList to parameters typed with the java.util.List interface and any other interface that extends java.util.Collection. Then these types are sent back to the client as mx.collections.ArrayCollection instances. If you require normal ActionScript Arrays sent back to the client, you must set the `legacy-collection` element to `true` in the `<serialization>` section of a channel-definition's properties; for more information, see "Providing legacy AMF serialization on a channel" on page 1372.

Converting data from Java to ActionScript

An object returned from a Java method is converted from Java to ActionScript. Flex also handles objects found within objects. Flex implicitly handles the Java data types in the following table:

Java type	ActionScript type (AMF 3)
java.lang.String	String
java.lang.Boolean, boolean	Boolean
java.lang.Integer	int If i < 0xF0000000 || i > 0x0FFFFFFF, the value is promoted to Number.
java.lang.Short	int If i < 0xF0000000 || i > 0x0FFFFFFF, the value is promoted to Number.
java.lang.Byte	int If i < 0xF0000000 || i > 0x0FFFFFFF, the value is promoted to Number.
java.lang.Byte[]	flash.utils.ByteArray
java.lang.Double	Number
java.lang.Long	Number
java.lang.Float	Number
java.lang.Character	String
java.lang.Character[]	String
java.util.Calendar	Date Dates are sent in the Coordinated Universal Time (UTC) time zone. Clients and servers must adjust time accordingly for time zones.

Java type	ActionScript type (AMF 3)
java.util.Date	Date Dates are sent in the UTC time zone. Clients and servers must adjust time accordingly for time zones.
java.lang.Object (other than previously listed types)	Typed Object Objects are serialized using Java Bean introspection rules. Fields that are static, transient, or nonpublic are excluded.
java.util.Collection	mx.collection.ArrayCollection
java.lang.Object[]	Array
java.util.Map	Object (untyped) In Flex 1.5, java.util.Map was sent as an associative or ECMA Array. This is no longer a recommended practice. You can enable legacy Map support to associative Arrays, but Adobe recommends against doing this. For more information, see "Providing legacy AMF serialization on a channel" on page 1372
java.util.Dictionary	Object (untyped)
org.w3c.dom.Document	XML object You can enable legacy XML support for the XMLDocument type on any channel defined in the services-config.xml file. For more information, see "Providing legacy AMF serialization on a channel" on page 1372.
null	null
Other classes that extend java.lang.Object	Object (typed) Objects are serialized using Java Bean introspection rules. Fields that are static, transient, or nonpublic are excluded.

For Java objects that Flex does not handle implicitly, values found in public getter/setter method pairs and public variables are passed to the client as properties on an Object. Private properties, constants, static properties, and read-only properties, and so on, are not serialized.

Flex uses the standard Java class, java.beans.Introspector, to get property descriptors for a Java bean class. That class contains the rules that Flex to determine property names on a corresponding ActionScript object.

In the ActionScript class, you use the `[RemoteClass(alias=" ")]` metadata tag to create an ActionScript object that maps directly to the Java object. The ActionScript class to which data is converted must be used or referenced in the MXML file. A good way to do this is by casting the result object, as the following example shows:

```
var result:MyClass = MyClass(event.result);
```

The class itself should use strongly typed references so that its dependencies are also linked.

The following examples shows the source code for an ActionScript class that uses the `[RemoteClass(alias=" ")]` metadata tag:

```
package samples.contact {
  [Bindable]
  [RemoteClass(alias="samples.contact.Contact")]
  public class Contact {
    public var contactId:int;

    public var firstName:String;

    public var lastName:String;

    public var address:String;

    public var city:String;

    public var state:String;

    public var zip:String;
  }
}
```

You can use the `[RemoteClass]` metadata tag without an alias if you do not map to a Java object on the server, but you do send back your object type from the server. Your ActionScript object is serialized to a Map object when it is sent to the server, but the object returned from the server to the clients is your original ActionScript type.

Providing legacy AMF serialization on a channel

To support legacy AMF type serialization used in earlier versions of Flex, you can set legacy serialization properties in channel definitions in the services-config.xml file. To use legacy serialization for a destination, you assign a legacy channel to that destination.

The following example shows a channel definition that uses legacy serialization.

```
<channel-definition id="my-legacy-amf"
  class="mx.messaging.channels.AMFChannel">
  <endpoint uri="http://{server.name}:8100/dev/messagebroker/amflegacy"
    class="flex.messaging.endpoints.AMFEndpoint"/>
```

```
    <properties>
      <serialization>
        <!-- ignore-property-errors is true by default. It determines
        if the endpoint should throw an error when an incoming
        client object has unexpected properties that cannot be
        set on the server object. -->
        <ignore-property-errors>true</ignore-property-errors>
        <!-- log-property-errors is false by default. When true,
        unexpected property errors are logged. -->
        <log-property-errors>false</log-property-errors>
        <legacy-collection>true</legacy-collection>
        <legacy-map>true</legacy-map>
        <legacy-xml>true</legacy-xml>
        <legacy-throwable>true</legacy-throwable>
      </serialization>
  </properties>
</channel-definition>
```

Converting from ActionScript to Schema and SOAP types for web services

Before sending SOAP requests that use the XML schema types in the following table, Flex requires the information to be appropriately represented in the corresponding ActionScript types. Complex types and the SOAP-encoded Array type are also supported, and these may comprise other complex types, arrays, or built-in XML schema types.

The following table contains type mappings between the XML schema types and ActionScript types for web services:

XML schema type	ActionScript type
ComplexType	Anonymous Object or any other typed object with expected properties defined
sequence	Array
boolean	Boolean
string	String
normalizedString	String
all number types	Number, int, uint
date	Date
dateTime	Date
time	Date
Base64Binary	flash.utils.ByteArray

XML schema type	ActionScript type
hexBinary	flash.utils.ByteArray
anyType (in WSDL)	Converts the ActionScript types to the following Schema types: • uint to unsignedInt • int to int • Number to double • Boolean to boolean • String to string • XMLDocument to schema • Date to dateTime • Array to array. If no type information is available, if no type information is available, the first item in the Array is used to determine the type

The following table contains type mappings between SOAP types and ActionScript types for web services:

SOAP type	ActionScript type
base64	flash.utils.ByteArray
number types	Number
boolean	Boolean
Array	Array
string	String

The following table contains type mappings between types that are specific to Apache Axis and ActionScript types for RPC-encoded web services:

Apache Axis type	ActionScript type
Map	Object
RowSet	Can only receive RowSets; can't send them.
Document	Can only receive Documents; can't send them.
Element	flash.xml.XMLNode

Using custom serialization

If the standard mechanisms for serializing and deserializing data between ActionScript on the client and Java on the server do not meet your needs, you can implement the ActionScript-based flash.utils.IExternalizable interface on the client and the corresponding Java-based java.io.Externalizable interface on the server to write your own serialization scheme.

A typical reason to use custom serialization is to avoid passing all of the properties of either the client-side or server-side representation of an object across the network tier. When you implement custom serialization, you can code your classes so that specific properties that are client-only or server-only are not passed over the wire. When you use the standard serialization scheme, all public properties are passed back and forth between the client and the server.

On the client side, the identity of a class that implements the flash.utils.IExternalizable interface is written in the serialization stream. The class serializes and reconstructs the state of its instances. The class implements the `writeExternal()` and `readExternal()` methods of the IExternalizable interface to get control over the contents and format of the serialization stream, but not the classname or type, for an object and its supertypes. These methods supersede the native AMF serialization behavior. These methods must be symmetrical with the supertype to save its state.

On the server side, a Java class that implements the java.io.Externalizable interface performs functionality that is analogous to an ActionScript class that implements the flash.utils.IExternalizable interface.

NOTE	You should not use types that implement the IExternalizable interface with the HTTPChannel if precise by-reference serialization is required. When you do this, references between recurring objects are lost and appear to be cloned at the endpoint.

The following example show the complete source code for the client (ActionScript) version of a Product class that maps to a Java-based Product class on the server. The client Product implements the IExternalizable interface, and the server Product implements the Externalizable interface.

```
// Product.as
package samples.externalizable {

import flash.utils.IExternalizable;
import flash.utils.IDataInput;
import flash.utils.IDataOutput;

[RemoteClass(alias="samples.externalizable.Product")]
public class Product implements IExternalizable {
  public function Product(name:String=null) {
    this.name = name;
```

```
    }

    public var id:int;
    public var name:String;
    public var properties:Object;
    public var price:Number;

    public function readExternal(input:IDataInput):void {
      name = input.readUTF();
      properties = input.readObject();
      price = input.readFloat();
    }

    public function writeExternal(output:IDataOutput):void {
      output.writeUTF(name);
      output.writeObject(properties);
      output.writeFloat(price);
    }
}
}
```

The client Product uses two kinds of serialization. It uses the standard serialization that is compatible with the java.io.Externalizable interface and AMF 3 serialization. The following example shows the `writeExternal()` method of the client Product, which uses both types of serialization:

```
public function writeExternal(output:IDataOutput):void {
  output.writeUTF(name);
  output.writeObject(properties);
  output.writeFloat(price);
}
```

As the following example shows, the `writeExternal()` method of the server Product is almost identical to the client version of this method:

```
public void writeExternal(ObjectOutput out) throws IOException {
  out.writeUTF(name);
  out.writeObject(properties);
  out.writeFloat(price);

}
```

In the client Product's `writeExternal()` method, the calls to the `flash.utils.IDataOutput.writeUTF()` method and the `flash.utils.IDataOutput.writeFloat()` method are examples of standard serialization methods that meet the specifications for the Java `java.io.DataInput.readUTF()` and `java.io.DataInput.readFloat()` methods for working with primitive types. These methods send the `name` property, which is a String, and the `price property`, which is a Float, to the server Product.

The only example of AMF 3 serialization in the client Product's `writeExternal()` method is the call to the `flash.utils.IDataOutput.writeObject()` method, which maps to the `java.io.ObjectInput.readObject()` method call in the server Product's `readExternal()` method. The `flash.utils.IDataOutput.writeObject()` method sends the `properties` property, which is an Object, to the server Product. This is possible because the AMFChannel endpoint has an implementation of the java.io.ObjectInput interface that expects data sent from the `writeObject()` method to be formatted as AMF 3.

In turn, when the `readObject()` method is called in the server Product's `readExternal()` method, it uses AMF 3 deserialization; this is why the ActionScript version of the `properties` value is assumed to be of type Map.

The following example shows the server Product's `readExternal()` method:

```
public void readExternal(ObjectInput in) throws IOException,
  ClassNotFoundException {
  // Read in the server properties from the client representation.
  name = in.readUTF();
  properties = (Map)in.readObject();
  price = in.readFloat();
  setInventoryId(lookupInventoryId(name, price));
  }
```

The client Product's `writeExternal()` method does not send the `id` property to the server during serialization because it is not useful to the server version of the Product object. Similarly, the server Product' `writeExternal()` method does not send the `inventoryId` property to the client because it is a server-specific property.

Notice that the names of a Product's properties are not sent during serialization in either direction. Because the state of the class is fixed and manageable, the properties are sent in a well-defined order without their names and the `readExternal()` method reads them in the appropriate order.

The following example shows the complete source of the server Product class:

```
// Product.java
package samples.externalizable;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Map;

/**
* This Externalizable class requires that clients sending and
* receiving instances of this type
* adhere to the data format required for serialization.
*/
```

```
public class Product implements Externalizable {
  private String inventoryId;
  public String name;
  public Map properties;
  public float price;

  public Product()
  {
  }

    /**
    * Local identity used to track third party inventory. This property is
    * not sent to the client because it is server-specific.
    * The identity must start with an 'X'.
    */
    public String getInventoryId() {
      return inventoryId;
    }

    public void setInventoryId(String inventoryId) {
      if (inventoryId != null && inventoryId.startsWith("X"))
      {
        this.inventoryId = inventoryId;
      }
      else
      {
        throw new IllegalArgumentException("3rd party product
        inventory identities must start with 'X'");
      }
    }

    /**
     * Deserializes the client state of an instance of ThirdPartyProxy
     * by reading in a UTF formatted String for the name, a UTF formatted
     * String for the description, and
     * a floating point integer (single precision) for the price.
     */
    public void readExternal(ObjectInput in) throws IOException,
      ClassNotFoundException {
      // Read in the server properties from the client representation.
      name = in.readUTF();
      properties = (Map)in.readObject();
      price = in.readFloat();
      setInventoryId(lookupInventoryId(name, price));
    }

    /**
     * Serializes the server state of an instance of ThirdPartyProxy
     * by sending a UTF-formatted String for the name, a UTF-formatted
     * String for the description, and a floating point
```

```
     * integer (single precision) for the price. Notice that the inventory
     * identifier is not sent to external clients.
     */
    public void writeExternal(ObjectOutput out) throws IOException {
      // Write out the client properties from the server representation
      out.writeUTF(name);
      out.writeObject(properties);
      out.writeFloat(price);
    }

    private static String lookupInventoryId(String name, float price) {
      String inventoryId = "X" + name + Math.rint(price);
      return inventoryId;
    }
}
```

Securing destinations

When a destination is not public, you can restrict access to a privileged group of users by applying a security constraint in a destination definition in the Flex services configuration file. A *security constraint* ensures that a user is authenticated, by using custom or basic authentication, before accessing the destination. By default, Flex Data Services security constraints use custom authentication. A security constraint can also require that a user is authorized against a user store before accessing a destination. A *user store* is a repository that contains security attributes of users.

Authentication is the process by which a user proves his or her identity to a system. *Authorization* is the process of determining what types of activities a user is permitted to perform in a system. After users are authenticated, they can be authorized to access specific resources.

You can declare a security constraint for a destination inline in a destination definition when the security constraint is used with only one destination. The following example shows a security constraint that is declared in a destination definition:

```
<service>
...
  <destination id="roDest">
...
    <security>
      <security-constraint>
        <auth-method>Custom</auth-method>
        <roles>
          <role>roDestUser</role>
        </roles>
      </security-constraint>
    </security>
```

```
  </destination>
...
</service>
```

You can also declare a security constraint globally. When several destinations use the same security settings, you should define one security constraint in the `security` section of the Flex services configuration file and refer to it in each destination. The following example shows a security constraint that is referenced in two destination definitions:

```
<service>
  <destination id="SecurePojo1">
...
    <security>
      <security-constraint ref="trusted"/>
    </security>
  </destination>
  <destination id="SecurePojo2">
...
    <security-constraint ref="trusted"/>
  </destination>
...
</service>
...
<security>
  <security-constraint id="trusted">
    <auth-method>Custom</auth-method>
    <roles>
      <role>trustedUsers</role>
    </roles>
  </security-constraint>
...
</security>
```

Passing credentials from client-side components

To send user credentials to a destination that uses a security constraint, you specify user name and password values as the parameters of the `setCredentials()` method of the RemoteObject, HTTPService, WebService, DataService, Publisher, or Consumer component that you are using in your application. You can remove credentials by calling the component's `logout()` method.

NOTE Calling the `setCredentials()` or `setRemoteCredentials()` method has no effect when a service component's `useProxy` property is set to `false`.

The following example shows ActionScript code for sending user name and password values from an HTTPService component to a destination:

```
import mx.rpc.http.HTTPService;

var employeeHTTP:HTTPService = new HTTPService();
employeeHTTP.destination = "SecureDest";
employeeHTTP.setCredentials("myUserName", "myPassword");
empoyeeHTTP.send({param1: 'foo'});
```

You can use the `setRemoteCredentials()` method to pass credentials to a remote HTTP service or web service that requires them. For example, when using an HTTPService component, you can pass credentials to a secured JSP page. The credentials are sent to the destination in message headers.

You can also call the `setRemoteCredentials()` method for Remoting Service destinations that are managed by an external service that requires user name and password authentication, such as a ColdFusion Component (CFC).

Passing remote credentials is distinct from passing user name and password credentials to satisfy a security constraint defined in the Flex services configuration file. However, the two types of credentials can be used in combination.

The following example shows ActionScript code for sending remote user name and remote password values to a destination. The destination configuration passes these credentials to the actual JSP page that requires them.

```
var employeeHTTP:mx.rpc.http.HTTPService = new HTTPService();
employeeHTTP.destination = "secureJSP";
employeeHTTP.setRemoteCredentials("myRemoteUserName", "myRemotePassword");
employeeHTTP.send({param1: 'foo'});
```

As an alternative to setting remote credentials on a client at run time, you can set remote credentials in `remote-username` and `remote-password` elements in the `<properties>` section of a server-side destination definition. The following example shows a destination that specifies these properties:

```
<destination id="samplesProxy">
  <channels>
    <channel ref="samples-amf"/>
  </channels>

  <properties>
    <url>
      http://someserver/SecureService.jsp
    </url>
    <remote-username>johndoe</remote-username>
    <remote-password>opensaysme</remote-password>
  </properties>
```

```
  </destination>
...
```

Basic authentication

Basic authentication relies on standard J2EE basic authentication from the web application container. To use this form of authentication, you secure a resource, such as a URL, in the web application's web.xml file. When you use basic authentication to secure access to destinations, you usually secure the endpoints of the channels that the destinations use in the web.xml file. You then configure the destination to access the secured resource in order to be challenged for a user name (principal) and password (credentials). The web browser performs the challenge, which happens independently of Flex. The web application container authenticates the user's credentials.

The following example shows a configuration for a secured channel endpoint in a web.xml file:

```
...
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected Channel</web-resource-name>

    <url-pattern>/messagebroker/amf</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>

  <auth-constraint>
    <role-name>sampleusers</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
</login-config>

<security-role>
  <role-name>sampleusers</role-name>
</security-role>
...
```

When users successfully log in, they remain logged in until the browser is closed. When applying a basic authentication security constraint to a destination, Flex checks that there is a currently authenticated principal before routing any messages to the destination. Even when there is a currently authenticated principal, you should perform custom authorization for the principal. You enable custom authorization by specifying roles in the destination's security constraint definition. The roles referenced in the web.xml file and in the security constraints that are defined in Flex Data Services configuration files are all defined in the application server's user store.

NOTE How you define users and roles is specific to your application server. For example, by default, you define Adobe JRun users and roles in the servers/server_name/SERVER-INF/jrun-users.xml file.

The following example shows a security constraint definition that specifies roles for authorization:

```
<security-constraint id="privileged-users">
  <auth-method>Basic</auth-method>
  <roles>
    <role>privilegedusers</role>
    <role>admins</role>
  </roles>
</security-constraint>
```

Custom authentication

As an alternative to basic authentication, you can use custom authentication and create a custom login form in MXML to match the appearance of your application.

For custom authentication, Flex uses a custom login adapter, known as a login command, to check a user's credentials and log a principal into the application server. A login command must implement the `flex.messaging.security.LoginCommand` API. You can register multiple login commands in the `<security>` section of the Flex services configuration file. The `server` attribute of the `login-command` element is used to perform a partial match against the value returned by the `servletConfig.getServletContext().getServerInfo()` method. The `server` value must be a case-insensitive match of an initial substring of this value or the entire string.

Flex Data Services includes login command implementations for Adobe JRun, Apache Tomcat, Oracle Application Server, BEA WebLogic, and IBM WebSphere. Use the TomcatLoginCommand class for either Tomcat or JBoss. You should enable only one login command at a time; comment out all others.

The following example shows a login command configured for JRun:

```
<security>
...
  <login-command class="flex.messaging.security.JRunLoginCommand"
    server="JRun"/>
<!--
  <login-command class="flex.messaging.security.TomcatLoginCommand"
    server="Tomcat"/>
  <login-command class="flex.messaging.security.OracleLoginCommand"
    server="Oracle"/>

  <login-command class="flex.messaging.security.WeblogicLoginCommand"
    server="Weblogic"/>
  <login-command class="flex.messaging.security.WebSphereLoginCommand"
    server="WebSphere"/>
       -->
...
</security>
```

You can use a login command without roles for custom authentication only. If you also want to use custom authorization, you must link the specified role references to roles that are defined in your application server's user store.

Configuring server-side service logging

You can perform server-side logging for Flex Data Services requests and responses. You configure server-side logging in the logging section of the Flex services configuration file. By default, output is sent to System.out. For information about client-side logging, see "Logging" on page 245 in *Building and Deploying Flex 2 Applications*.

You set the logging level to one of the following available levels:

- `all`
- `debug`
- `info`
- `warn`
- `error`
- `none`

In the filter pattern elements, you can specify the categories of information to log. In the `class` attribute of the `target` element, you can specify `flex.messaging.log.ConsoleTarget` to log messages to the standard output or the `flex.messaging.log.ServletLogTarget` to log messages to use your application server's default logging mechanism for servlets.

The following example shows a logging configuration that uses the Debug logging level:

```
<logging>

<!-- You may also use flex.messaging.log.ServletLogTarget -->
  <target class="flex.messaging.log.ConsoleTarget" level="Debug">
    <properties>
      <prefix>[Flex]</prefix>
      <includeDate>false</includeDate>
      <includeTime>false</includeTime>
      <includeLevel>false</includeLevel>
      <includeCategory>false</includeCategory>
    </properties>
    <filters>
      <pattern>Endpoint</pattern>
      <!--<pattern>Service.*</pattern>-->
      <!--<pattern>Message.*</pattern>-->
    </filters>
  </target>
</logging>
```

In a production environment, Adobe recommends that you set the logging level to `warn` so both warnings and errors are displayed. If there are warnings that occur frequently that you want to ignore, you can change the level to `error` so that only errors are displayed.

The following table describes the logging levels:

Logging level	Description
`all`	Designates that all messages should be logged.
`debug`	`DEBUG` messages indicate internal Flex activities. Select the `debug` logging level to include `Debug`, `Info`, `Warn`, and `Error` log messages in your log files.
`info`	`INFO` messages indicate general information to the developer or administrator. Select the `Info` logging level to include `Info`, `Warn`, and `Error` log messages in your log files.

Logging level	Description
warn	Warn messages indicate that Flex encountered a problem with the application, but it does not stop running. Select the warn logging level to include warn and error log messages in your log files.
error	Error messages indicate when a critical service is not available or a situation has occurred that restricts use of the application. Select the Error logging level to include Error log messages in your log files.
none	Designates that no messages should be logged.

You can use the following matching patterns as the values of pattern elements:

- Configuration
- DataService.General
- DataService.Hibernate
- DataService.Transaction
- Endpoint.*
- Endpoint.AMF
- Endpoint.HTTP
- Endpoint.RTMP
- Endpoint.Deserialization
- Endpoint.General
- Message.*
- Message.Command.*
- Message.Command.*operation-name* where *operation-name* is one of the following:
  subscribe, unsubscribe, poll, poll_interval, client_sync, server_ping,client_ping, cluster_request, login, logout
- Message.General
- Message.Data.*
- Message.Data.*operation-name* where *operation-name* is one of the following:
  create, fill get, update, delete, batched, multi_batch, transacted, page, count, get_or_create, create_and_sequence, get_sequence_id, association_add, association_remove, fillids, refresh_fill, update_collection
- Message.RPC
- MessageSelector
- Resource

- Service.*
- Service.Cluster
- Service.HTTP
- Service.Message
- Service.Message.JMS (logs a warning if a durable JMS subscriber can't be unsubscribed successfully.)
- Service.Remoting
- Security

NOTE

How you define users and roles is specific to your application server. For example, by default, you define Adobe JRun users and roles in the servers/server_name/SERVER-INF/jrun-users.xml file.

Working with session data

Flex Data Services provides the following classes for working with session data. These classes are included in the public Flex Data Services Javadoc documentation.

- flex.messaging.FlexContext
- flex.messaging.FlexSession
- flex.messaging.FlexSessionListener
- flex.messaging.FlexSessionAttributeListener
- flex.messaging.FlexSessionBindingEvent
- flex.messaging.FlexSessionBindingListener

The FlexContext class is useful for getting access to the session and the HTTP pieces of the session, such as the HTTP servlet request and response. This lets you access HTTP data when you use a Flex application in the context of a larger web application where other classes, such as JSPs or Struts actions, may have stored information.

The FlexSession class provides access to an ID and also provides setAttribute and getAttribute functionality. This is useful for storing data on the server that doesn't have to go back to the client. However, Flex Session is not cluster-aware; if a client connects to a different server in the cluster, the client receives a new FlexSession. Nothing stored in the FlexSession attributes is persisted for clustering purposes. You should consider the following when using FlexSession:

- When using an RTMP channel, the FlexSession does not remain after a browser refresh, but it does remain when using an AMF or HTTP channel.

- When using an RTMP channel, a FlexSession's attributes are not shared with other channels, but they are shared when using an AMF or HTTP channel.
- When using an RTMP channel with authentication, you must log in again after a browser refresh, but this is not necessary when using an AMF or HTTP channel.
- When using an RTMP channel with authentication, you must log in again after switching to an AMF or an HTTP channel.

The FlexSessionListener class is useful for monitoring who is connected. You add a listener by using the static method to track new connections being made. You receive a reference to the session that was added. Each session can then report when it is destroyed to those same listeners. You can use this for monitoring connections that close, and also to clean up resources.

Using software clustering

The Flex Data Services software clustering feature handles failover when using stateful services and when using non-HTTP channels, such as the RTMP channel. When a Flex client subscribes to a particular service on one host in a subnet, and the host fails, the client has the ability to direct further messages to another host in the same subnet. This feature is available for all of the service types defined in the Flex services configuration file. For the Message Service and the Data Management Service, both failover and replication of an application's messaging state is supported. For the Remoting Service and Proxy Service, failover only is supported.

The most common form of clustering for Flex Data Services does not involve software clustering. Instead, a set of load balancers, usually in the form of hardware, is positioned in front of the HTTP servers. These load balancers direct requests to individual HTTP servers and pin client requests from the same client to that HTTP server. This form of clustering is possible with Flex without any feature implementation. It can also work in conjunction with the software clustering feature.

Processing messages

In addition to providing client failover, cluster nodes may have to process a message on all nodes. This is usually the process when no back-end resource is available to coordinate the common cluster state. When no shared back end is available, a cluster node directs the other node to reprocess and broadcast a message. When a shared back end is available, a node directs the other nodes to broadcast the message to connected clients, but it does not direct the other nodes to reprocess the message.

Defining and referencing a cluster

You define software clusters in the `clusters` section of the Flex services configuration file. You can define multiple clusters. Each cluster definition must have an `id` and a path to a JGroups properties file. JGroups is the open source software that Flex Data Services uses to provide software clustering. For information about JGroups, see www.jgroups.org/javagroupsnew/docs/index.html.

The jgroups.jar file and the jgroups-*.xml properties files are located in the resources/clustering folder of the Flex Data Services installation. You must copy the jgroups.jar file to the WEB-INF/lib and copy the jgroups-*.xml files to the WEB-INF/flex directory before you define clusters.

You reference a named cluster in the `<network>` section of the destinations that you want to include in the cluster. The channel that a clustered destination points to cannot contain any tokens in the URI specified as its endpoint.

The following example shows the configuration of a cluster in the services-config.xml file:

```
<?xml version="1.0"?>
<services-config>
...
  <clusters>
        <cluster id="MyCluster" properties="jgroups-udp.xml"/>
        <cluster id="AnotherCluster" properties="jgroups-tcp.xml"/>
    </clusters>
...
</services-config>
```

The following example shows a reference to a cluster in a service destination. Because the `shared back-end` property is set to `true`, a cluster node directs the other nodes to broadcast messages to connected clients, but it does not direct the other nodes to reprocess the message. The `shared-backend` attribute is optional and is relevant only for Data Management Service destinations.

```
<destination id="MyDestination">
...
  <properties>
    <network>
      <cluster ref="default-cluster" shared-backend="true"/>
    </network>
  </properties>
...
</destination>
```

Managing services

Flex Data Services uses Java Management Beans (MBeans) to provide run-time monitoring and management of the services configured in the Flex services configuration file. The run-time monitoring and management console is an example of a Flex client application that provides access to the run-time MBeans. The application calls a Remoting Service destination, which is a Java class that makes calls to the MBeans. The console is in the flex-admin web application; you run it by opening http://*server:port*/flex-admin when the web application is running, where *server:port* contains your server and port names.

NOTE

The run-time monitoring and management console exposes administrative functionality without authorization checks. You should deploy the flex-admin web application to the same application server that your Flex web application is deployed to, and you should lock down the flex-admin web application by using J2EE security or some other means to protect access to it.

MBean creation and registration

The APIs exposed by the run-time MBeans do not affect or interact with the Flex Data Services configuration files. You can use them for operations such as ending a stale connection or monitoring message throttling, but you cannot use them for operations such as registering a new service or altering the settings for an existing server component.

The run-time MBeans are instantiated and registered with the local MBean server by their corresponding managed resource. For example, when a MessageBroker is instantiated, it creates and registers a corresponding MessageBrokerControlMBean with the MBean server. The underlying resource sets the attributes for the run-time MBeans and they are exposed in a read-only manner by the MBean API. In some cases, an MBean can poll its underlying resource for an attribute value.

MBean naming conventions

The run-time MBean model starts at the MessageBrokerControlMBean. You can traverse the model by using attributes on MBeans that reference other MBeans. For example, to access an EndpointControlMBean, you start at the MessageBrokerControlMBean and get the `Endpoints` attribute. This attribute contains the ObjectNames of all endpoints that are registered with the management broker. This allows any management client to generate a single ObjectName for the root MessageBrokerControlMBean, and from there, navigate to and manage any of the other MBeans in the system without having their ObjectNames.

The run-time MBean model is organized hierarchically, however, each registered Mean in the system has an MBean ObjectName and can be fetched or queried directly if necessary. The run-time MBeans follow ObjectName conventions to simplify registration, lookup, and querying. An ObjectName for an MBean instance is defined as follows:

```
{domain}:{key}={value}[,{keyN}={valueN}]*
```

You can provide any number of additional key-value pairs to uniquely identify the MBean instance.

All of the run-time MBeans belong to the flex.run-time domain. If an application name is available, it is also included in the domain as follows: `flex.runtime.application-name`

Each of the run-time MBean ObjectNames contains the following keys:

Key	Description
type	Short type name of the resource managed by the MBean. The MessageBrokerControlMBean manages the flex.messaging.MessageBroker, so its type is MessageBroker.
id	The `id` value of the resource managed by the MBean. If no `name` or `id` is available on the resource, an `id` is created according to this strategy: `id = {type} + N` Where *N* is a numeric increment for instances of this type.

An ObjectName can also contain additional optional keys.

The run-time MBeans are documented in the public Flex Data Services Javadoc documentation. The Javadoc also includes documentation for the flex.management.jmx.MBeanServerGateway class, which the run-time monitoring and management console uses as a Remoting Service destination.

Creating a custom MBean for a custom ServiceAdapter class

If you choose to write a custom MBean to expose metrics or management APIs for a custom ServiceAdapter, the ServiceAdapter class defines the following method to let you wire up your corresponding MBean:

```
public void setupAdapterControl(Destination destination);
```

Your custom ServiceAdapter control MBean should extend flex.management.runtime.messaging.services.ServiceAdapterControl. Your MBean must implement your custom MBean interface, which in turn must extend flex.management.runtime.messaging.services.ServiceAdapterControlMBean. This allows your custom MBean to be added into the hierarchy of core Flex Data Services MBeans.

The code in the following example shows how to implement a `setupAdapterControl()` method in your custom ServiceAdapter, where `controller` is an instance variable of type CustomAdapterControl (your custom MBean implementation class):

```
public void setupAdapterControl(Destination destination) {
  controller = new CustomAdapterControl(getId(), this,
  destination.getControl());
  controller.register();
  setControl(controller);
}
```

Your custom adapter can update metrics stored in its corresponding MBean by invoking methods or updating properties of `controller`, the MBean instance. Your custom MBean is passed a reference to your custom adapter in its constructor. You can access this reference in your custom MBean by using the `protected ServiceAdapter serviceAdapter` instance variable. This allows your MBean to query its corresponding service adapter for its state or invoke methods on it.

Using custom error handling

For use cases where you return additional information to the client as part of a message exception, you can use the `extendedData` property of the flex.messaging.MessageException class. This property is a HashMap, and provides a flexible way to return additional data to the client when a failure occurs. The Javadoc documentation in the docs directory of your Flex Data Services installation includes documentation for the flex.messaging.MessageException class.

NOTE	Flex Data Services serialization provides bean serialization of any Throwable type. This gives you the option of throwing your own Throwable exceptions with getters for the properties that you want to send to the client.

The following example shows a Java test class that adds extra data to an exception:

```
package errorhandling;

import java.util.HashMap;
import java.util.Map;
import flex.messaging.MessageException;

public class TestException {

public String generateMessageExceptionWithExtendedData(String extraData)
{
  MessageException me = new MessageException("Testing extendedData.");
  Map extendedData = new HashMap();
```

```
    // The method that invokes this expects an "extraData" slot in
    // this map.
    extendedData.put("extraData", extraData);
    me.setExtendedData(extendedData);
    me.setCode("999");
    me.setDetails("Some custom details.");
    throw me;
    }

}
```

The following example shows an ActionScript method that generates an exception with extra data:

```
<?xml version="1.0"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="run()">

  <mx:RemoteObject
    destination="myDest"
    id="myException"
    fault="onServiceFault(event)"
/>
  <mx:Script>
  <![CDATA[
    import mx.rpc.events.*;
    import mx.messaging.messages.*;"

    public var data : String = "Extra data.";
    public var actualData : String;

    public function onServiceFault(event:FaultEvent):void {
      actualData = ErrorMessage(event.message).extendedData.extraData;
    }

    public function run_exception():void {
      var call : Object =
        myException.generateMessageExceptionWithExtendedData(data);
    }

    public function run():void {
      run_exception();
    }
  ]]>
  </mx:Script>
</mx:Application>
```

About Data Services class loading

Flex Data Services for the most part loads its classes in the same way as standard J2EE web applications using the default class loader that the application server provides. The one exception to this rule is for the web-tier compiler. It uses a bootstrap classloader and loads JAR files from the value of the context-param in web.xml with the name `flex.class.path`. By default this is set to find the flex web-tier compiler jar files in the WEB-INF/flex/jars directory. This is an attempt to isolate the web-tier classes from potential conflicts in the web application classloader; however, the actual services of Flex Data Services do not use that classloader. As is typical with J2EE web applications, you might encounter issues when trying to use your own versions of JAR files that conflict with those that already exist in the application server source path. When troubleshooting any issues, always first ensure that your code runs in an application without Flex Data Services.

If you do not plan to use the web-tier compiler, you can use the data services without it. For more information, see "Using data services without the web-tier compiler" on page 1395.

Web-tier compiler class loading

In the web.xml file, the FlexMxmlServlet servlet class is the flex.bootstrap.BootstrapServlet class, with an init-param that specifies flex2.server.j2ee.MxmlServlet as the servlet that is loaded to handle MXML compilation. The compiler exists in the mxmlc.jar and asc.jar files located in WEB-INF/flex/jars, which is accessible to the Flex web-tier classloader rather than to the web application classloader.

The following example shows the FlexMxmlServlet definition in the web.xml file:

```
<servlet>
  <servlet-name>FlexMxmlServlet</servlet-name>
  <display-name>MXML Processor</display-name>
  <description>Servlet wrapper for the Mxml Compiler</description>
  <servlet-class>flex.bootstrap.BootstrapServlet</servlet-class>
  <init-param>
    <param-name>servlet.class</param-name>
    <param-value>flex2.server.j2ee.MxmlServlet</param-value>
  </init-param>
  <init-param>
    <param-name>webtier.configuration.file</param-name>
    <param-value>/WEB-INF/flex/flex-webtier-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The BootstrapServlet uses its own class path, which is specified in a context-param called `flex.class.path` in the web.xml file. The following example specifies that the BootstrapServlet (which loads the web-tier compiler) looks for classes in two directories the flex/hotfixes and the flex/jars directories:

```
<context-param>
  <param-name>flex.class.path</param-name>
  <param-value>/WEB-INF/flex/hotfixes,/WEB-INF/flex/jars</param-value>
</context-param>
```

Data services class loading

The MessageBrokerServlet creates the Flex Data Services message broker. This servlet starts all of the services and endpoints, including the RTMP server. It is loaded by the standard class loader used by the application server to load classes for that web application (it does not use the BootstrapServlet which uses a special bootstrap class loader). As with standard J2EE web applications, Flex Data Services loads classes from the WEB-INF/lib directory using the web application classloader. There is some classloader context switching when the MXML compiler or web-tier cache is accessed, but there is none when the Flex Data Services services are accessed.

The following example shows the MessageBrokerServlet definition in the web.xml file:

```
<servlet>
  <servlet-name>MessageBrokerServlet</servlet-name>
  <display-name>MessageBrokerServlet</display-name>
  <servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>
  <init-param>
    <param-name>services.configuration.file</param-name>
    <param-value>/WEB-INF/flex/services-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Using data services without the web-tier compiler

You can use the data services without the web-tier compiler. If you do not use the web-tier compiler, the only files you must have in the WEB-INF/flex directory are the license.properties file and the services-config.xml, proxy-config.xml, messaging-config.xml, and data-management-config.xml files.

You can remove the flex-bootstrap.jar file from WEB-INF/lib directory if you do not use the MXML compiler. You should also remove the servlet definition for the FlexMxmlServlet from the web.xml file.

Using the factory mechanism

Flex Remoting Service destinations and Data Management Service destinations both use Java classes that you write to integrate with Flex clients. By default, Flex Data Services creates these instances. If they are application-scoped components, they are stored in the ServletContext attribute space using the destination's name as the attribute name. If you use session-scoped components, the components are stored in the FlexSession, also using the destination name as the attribute. If you specify an `attribute-id` element in the destination, you can control which attribute the component is stored in; this lets more than one destination share the same instance.

The following examples shows a destination definition that contains an `attribute-id` element:

```
<destination id="WeatherService">
  <properties>
    <source>weather.WeatherService</source>
    <scope>application</scope>
    <attribute-id>MyWeatherService</attribute-id>
  </properties>
</destination>
```

In this example, Flex creates an instance of the class weather.WeatherService and stores it in the ServletContext's set of attributes with the name MyWeatherService. If you define a different destination with the same `attribute-id` value and the same Java class, Flex uses the same component instance.

Flex Data Services provides a factory mechanism that lets you plug in your own component creation and maintenance system to Flex to allow it to integrate with systems like EJB and Spring, which store components in their own namespace. You provide a class that implements the flex.messaging.FlexFactory interface. This class is used to create a FactoryInstance that corresponds to a component configured for a specific destination. It then uses the FactoryInstance to look up the specific component to use for a given request. The FlexFactory implementation can access configuration attributes from a Flex Data Services configuration file and also can access FlexSession and ServletContext objects. For more information, see the documentation for the FlexFactory class in the public Flex Data Services Javadoc documentation.

After you implement a FlexFactory class, you can configure it to be available to destinations by placing a `factory` element in the `factories` element in the services-config.xml file, as the following example shows. A single FlexFactory instance is created for each Flex-enabled web application. This instance can have its own configuration properties, although in this example, there are no required properties to configure the factory.

```
<factories>
  <factory id="spring" class="flex.samples.factories.SpringFactory" />
</factories>
```

Flex creates one FlexFactory instance for each `factory` element you specify. Flex uses this one global FlexFactory implementation for each destination that specifies that factory by its ID using a `factory` element in the `properties` section of the destination definition. For example, your remoting-config.xml file could have a destination similar to the following one:

```
<destination id="WeatherService">
  <properties>
    <factory>spring</factory>
    <source>weatherBean</source>
  </properties>
</destination>
```

When the `factory` element is encountered at start up, Flex Data Services calls the `FlexFactory.createFactoryInstance()` method. That method gets the `source` value and any other attributes it expects in the configuration. Any attributes that you access from the ConfigMap parameter are marked as expected and do not cause a configuration error, so you can extend the default Flex configuration in this manner. When Flex needs an instance of the component for this destination, it calls the `FactoryInstance.lookup()` method to retrieve the individual instance for the destination.

Optionally, factory instances can take additional attributes. There are two places you can do this. When you define the factory initially, you can provide extra attributes as part of the factory definition. When you define an instance of that factory, you can also add your own attributes to the destination definition to be used in creating the factory instance.

The boldface text in the following example shows an attribute added to the factory definition:

```
<factories>
  <factory id="myFactoryId" class="myPackage.MyFlexFactory">
    <properties>
      <myfactoryattributename>
        myfactoryattributevalue
      </myfactoryattributename>
    </properties>
  </factory>
</factories>
```

An example of where you would use this type of configuration is when you are integrating with the Spring Framework Java application framework, to provide the Spring factory with a default path for initializing the Spring context used to look up all components for that factory. In the class that implements FlexFactory, you would include a call to retrieve the values of the `myfactoryattributename` from the `configMap` parameter to the `initialize()` method in the FlexFactory interface, as the following example shows:

```
public void initialize(String id, ConfigMap configMap){
  System.out.println("**** MyFactory initialized with: " +
  configMap.getPropertyAsString("myfactoryattributename", "not set"));
}
```

The `initialize()` method in the previous example retrieves a string value where the first parameter is the name of the attribute, and the second parameter is the default value to use if that value is not set. For more information about the various calls to retrieve properties in the config map, see the documentation for the flex.messaging.config.ConfigMap class in the public Flex Data Services Javadoc documentation.

Each factory instance can add configuration attributes that are used when that factory instance is defined, as the following example shows:

```
<destination id="myDestination">
  <properties>
    <source>mypackage.MyRemoteClass</source>
    <factory>myFactoryId</factory>
    <myfactoryinstanceattribute>
      myfoobar2value
    </myfactoryinstanceattribute>
  </properties>
</destination>
```

In the `createFactoryInstance()` method as part of the FlexFactory implementation, you access the attribute for that instance of the factory, as the following example shows:

```
public FactoryInstance createFactoryInstance(String id, ConfigMap
  properties) {
  System.out.println("**** MyFactoryInstance instance initialized with
  myfactoryinstanceattribute=" +
  properties.getPropertyAsString("myfactoryinstanceattribute", "not
  set"));
….
}
```

CHAPTER 44

Understanding RPC Components

44

This topic introduces Adobe Flex remote procedure call (RPC) components. Flex RPC components are based on a service-oriented architecture (SOA). RPC components let you interact with server-side RPC services to provide data to your applications.

You can access data through HTTP GET or POST (HTTP services), SOAP (web services), or Java objects (remote object services). Another common name for an HTTP service is a REST-style web service. REST stands for Representational State Transfer and is an architectural style for distributed hypermedia systems. For more information about REST, see www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

In a typical Flex application, an RPC component sends data as input to one or more RPC services. When an RPC service executes, it returns its results data to the RPC component that made the request.

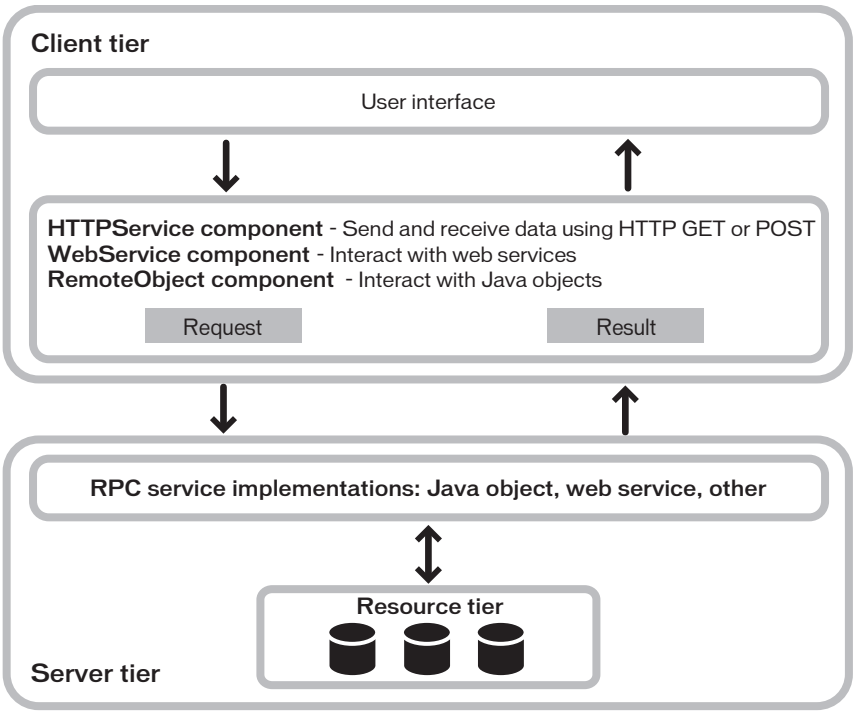
Contents

About RPC components

You can use Flex RPC components in MXML or ActionScript to work with three types of RPC services: remote object services, web services, and HTTP services. This section briefly describes the different types of RPC components and when to use them.

When you use Flex framework without Adobe Flex Data Services, you contact RPC services directly. When you use Flex Data Services, you can also contact *destinations*, which are RPC services that have corresponding server-side configurations that provide server-proxied access.

The following diagram provides a simplified view of how RPC components interact with RPC services:



The following list describes some key considerations when you are creating an application that must access an RPC service:

1. What is the best type of service to use? For more information, see "RemoteObject components" on page 1401, "WebService components" on page 1401, and "HTTPService components" on page 1402.
2. Should you contact a service directly or go through a destination? For more information, see "Declaring an RPC component" on page 1407.
3. What is the best way to pass data to a service? For more information, see "Calling a service" on page 1413.
4. How do you want to handle data results from a service? For more information, see "Handling service results" on page 1428.
5. How can you debug your code that uses RPC components? For more information, see "Configuring server-side service logging" on page 1384.

6. What security measures can or should you implement? For more information, see "Securing destinations" on page 1379.

RemoteObject components

RemoteObject components let you access the methods of server-side objects, such as Java objects, without manually configuring the objects as web services. Unlike WebService and HTTPService components, you can only use RemoteObject components in conjunction with Flex Data Services or Macromedia ColdFusion MX 7.0.2 with the Remoting Update. You configure the objects that you want to access as Remoting Service destinations in the Flex services configuration file that is part of Flex Data Services. You can use RemoteObject components in MXML or ActionScript.

You can use a RemoteObject component instead of a WebService component when objects are not already published as web services, web services are not used in your environment, or you would rather use Java objects than web services. You can use a RemoteObject component to connect to a local Java object that is in the Flex Data Services or ColdFusion MX 7.0.2 web application source path.

When you use a RemoteObject tag, data is passed between your application and the server-side object in the binary Action Message Format (AMF) format.

For more information about using RemoteObject components, see Chapter 45, "Using RPC Components," on page 1407.

WebService components

WebService components let you access *web services*, which are software modules with methods, commonly referred to as *operations*; web service interfaces are defined by using XML. Web services provide a standards-compliant way for software modules that are running on a variety of platforms to interact with each other. For more information about web services, see the web services section of the World Wide Web Consortium's website at www.w3.org/2002/ws/.

Flex applications can interact with web services that define their interfaces in a Web Services Description Language (WSDL) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages.

Flex supports WSDL 1.1, which is described at www.w3.org/TR/wsdl. Flex supports both RPC-encoded and document-literal web services.

Flex applications support web service requests and results that are formatted as SOAP messages and are transported over HTTP. SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

You can use a WebService component to connect to a SOAP-compliant web service when web services are an established standard in your environment. WebService components are also useful for objects that are within an enterprise environment, but not necessarily available to the Flex web application's source path.

For more information about using WebService components, see Chapter 45, "Using RPC Components," on page 1407.

HTTPService components

HTTPService components let you send HTTP GET, POST, HEAD, OPTIONS, PUT, TRACE or DELETE requests, and include data from HTTP responses in a Flex application. Flex does not support mulitpart form POSTs.

An HTTP service can be any HTTP URI that accepts HTTP requests and sends responses. Another common name for this type of service is a REST-style web service. REST stands for Representational State Transfer and is an architectural style for distributed hypermedia systems. For more information about REST, see www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

HTTPService components are a good option when you cannot expose the same functionality as a SOAP web service or a remote object service. For example, you can use HTTPService components to interact with JavaServer Pages (JSPs), servlets, and ASP pages that are not available as web services or Remoting Service destinations.

You can use an HTTPService component for CGI-like interaction in which you use HTTP GET, POST, HEAD, OPTIONS, PUT, TRACE, or DELETE to send a request to a specified URI. When you call the HTTPService object's `send()` method, it makes an HTTP request to the specified URI, and an HTTP response is returned. Optionally, you can pass arguments to the specified URI.

Example: An RPC component

The following example shows MXML code for a RemoteObject component that connects to a Remoting Service destination, sends a request to the data source in the `click` event of a Button control, and displays the result data in the `text` property of a TextArea control. The value in the curly braces ({ }) in the TextArea control is a binding expression that copies service results data into the `text` property of the TextArea control:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCIntroExample1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Connect to a service destination.-->
    <mx:RemoteObject id="remoteService" destination="myRO"/>

    <!-- Provide input data for calling the service. -->
    <mx:TextInput id="inputText"/>

    <!-- Call the web service, use the text in a TextInput control as input
data.-->
    <mx:Button click="remoteService.getData(inputText.text)"/>

    <!-- Display results data in the user interface. -->
    <mx:TextArea text="{remoteService.getData.lastResult.prop1}"/>
</mx:Application>
```

For more information, see Chapter 45, “Using RPC Components,” on page 1407.

Comparing the Flex RPC services feature to other technologies

The way that Flex works with data sources and data is different from other web application environments, such as JSP, ASP, and ColdFusion. Data access in Flex applications also differs significantly from data access in applications that are created in Flash Professional. This section describes some of the differences.

Client-side processing and server-side processing

Unlike a set of HTML templates created using JSPs and servlets, ASP, or CFML, the files in a Flex application are compiled into a binary SWF file that is sent to the client. When a Flex application makes a request to an external service, the SWF file is not recompiled and no page refresh is required.

The following example shows MXML code for calling a web service directly (without going through the Proxy Service). The default value of the `WebService.useProxy` property is `false`, so it is not declared in the tag. When a user clicks the Button control, client-side code calls the web service, and result data is returned into the binary SWF file without a page refresh. The result data is then available to use as dynamic content within the application. The following example shows a `wsdl` property on a WebService component, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCIntroExample2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <!-- Declare a WebService component (the specified WSDL URL is not
functional). -->
    <mx:WebService id="WeatherService"
        wsdl="/ws/WeatherService?wsdl"/>

    <mx:Button label="Get Weather"
        click="WeatherService.GetWeather(input.text);"/>

    <mx:TextInput id="input"/>
</mx:Application>
```

The following example shows JSP code for calling a web service using a JSP custom tag. When a user requests this JSP, the web service request is made on the server instead of on the client, and the result is used to generate content in the HTML page. The application server regenerates the entire HTML page before sending it back to the user's web browser.

```
<%@ taglib prefix="web" uri="webservicetag" %>

<% String str1="BRL";
String str2="USD";%>

<!-- Call the web service. -->
<web:invoke
  url="http://www.itfinity.net:8008/soap/exrates/default.asp"
  namespace="http://www.itfinity.net/soap/exrates/exrates.xsd"
  operation="GetRate"
  resulttype="double"
  result="myresult">
  <web:param name="fromCurr" value="<%=str1%>"/>
  <web:param name="ToCurr" value="<%=str2%>"/>
</web:invoke>

<!-- Display the web service result. -->
<%= pageContext.getAttribute("myresult") %>
```

Data source access

Another difference between Flex and other web application technologies is that you never communicate directly with a data source in Flex. You use a Flex service component to connect to a server-side service that interacts with the data source.

The following example shows one way to access a data source directly in a ColdFusion page:

```
...
<CFQUERY DATASOURCE="Dsn"
  NAME="myQuery">
  SELECT * FROM table
</CFQUERY>
...
```

To get similar functionality in Flex, you use an HTTPService, a WebService, or a RemoteObject component to call a server-side object that returns results from a data source.

Flash Professional data management

Flash Professional and Flex 2 provide different data management architectures. These architectures were developed to meet the needs of the respective authoring environments and user communities. Flash Professional provides a set of data components designed for use in the Flash Professional authoring environment. Although some of the functionality of these components overlaps with features found in Flex, they are not based on the same architecture.

Flash Professional also has its own data-binding feature that works in conjunction with the Flash Professional components, and is a completely different feature than Flex data binding.

CHAPTER 45

Using RPC Components

45

Adobe Flex 2 supports a service-oriented architecture in which a Flex application interacts with remote data sources by calling several types of services and receiving responses from the services. In a typical Flex application, client-side RPC components send asynchronous requests to remote services, which return result data to the client-side components.

This topic describes how to use Flex RPC components. For introductory information about RPC components, see Chapter 44, "Understanding RPC Components," on page 1399.

Contents

Declaring an RPC component

You can declare RPC components in MXML or ActionScript to connect to RPC services. If you are not using Adobe Flex Data Services, you can use a WebService or HTTPService component to contact an RPC service directly.

NOTE

To use RemoteObject components, Flex Data Services or Macromedia ColdFusion MX 7.0.2 with the Remoting Update is required.

If you are using Flex Data Services, you have the option of connecting to RPC services directly or connecting to destinations defined in the services-config.xml file or a file that it includes by reference. A destination definition is a named service configuration that provides server-proxied access to an RPC service. A destination is the actual service or object that you want to call. A destination for a RemoteObject component is a Java object. A destination for an HTTPService component is a JSP page or another resource accessed over HTTP. A destination for a WebService component is a SOAP-compliant web service.

Destination definitions provide centralized administration of RPC services. They also enable you to use basic or custom authentication to secure access to destinations. You can choose from several different transport channels, including secure channels, for sending data to and from destinations. Additionally, you can use the server-side logging feature to log RPC service traffic.

Optionally, you can use an HTTPService component or a WebService component to connect to a default destination.

This section discusses the basics of declaring a service connection in MXML and configuring a destination. For information about calling a service in MXML and ActionScript, see "Calling a service" on page 1413. For information about configuring destinations, see Chapter 46, "Configuring RPC Services," on page 1451.

Using a service without server-side configuration

You can use RPC components to connect to HTTP services and web services without configuring Flex Data Services destinations. To do so, you set an HTTPService component's `url` property or a WebService component's `wsdl` property instead of setting the component's `destination` property.

When an RPC component's `useProxy` property is set to `false`, the default value, the component communicates directly with the RPC service based on the `url` or `wsdl` property value. If you are not using Flex Data Services, the `useProxy` property value must be `false`. Connecting to a service in this manner requires that at least one of the following is true:

- The RPC service is in the same domain as your Flex application.
- A crossdomain.xml (cross-domain policy) file that allows access from your application's domain is installed on the web server hosting the RPC service. For more information about cross-domain policy, see Chapter 4, "Applying Flex Security" in *Building and Deploying Flex 2 Applications*.

When you set an HTTPService or WebService component's `useProxy` property to `true` and set the `url` or `wsdl` property and do not set the `destination` property, the component uses a default destination that is configured in the `<proxy-service>` section of the services-config.xml file or a file that it includes by reference, such as the proxy-config.xml file. This requires Flex Data Services, and the `defaultHTTP` or `defaultHTTPS` destination in the configuration file must contain a URL pattern that matches the value of your component's `url` or `wsdl` property, respectively. The defaultHTTPS destination is used when your `url` or `wsdl` property specifies an HTTPS URL.

The following examples show MXML tags for declaring HTTPService and WebService components that directly reference RPC services. The `id` property is required for calling the services and handling service results. In these examples, the `useProxy` property is not set in the tags; the components use the default `useProxy` value of `false` and contact the services directly, as is required when you are not using Flex Data Services.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCNoServer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:HTTPService
        id="yahoo_web_search"
        url="http://api.search.yahoo.com/WebSearchService/V1/webSearch"/>

    <mx:WebService
        id="macr_news"
        wsdl="http://ws.invesbot.com/companysearch.asmx?wsdl"/>
</mx:Application>
```

The following examples show ActionScript code for declaring HTTPService and WebService components with no corresponding server-side configuration. In these examples, the `useProxy` property is not set explicitly; the components use the default `useProxy` value of `false` and contact the services directly, as is required when you are not using Flex Data Services. You would place the code for declaring the components in ActionScript methods:

NOTE	You must use the Flex proxy, which is part of Flex Data Services, to ensure that HTTP status codes are returned correctly from HTTP services and web services. On web browsers, when a service returns any status code other than 200, Adobe Flash Player cannot read the body of the response. If the status code is 500 and the body contains a fault, there is no way to get to the fault. The proxy works around this issue by forcing the status code for faults to 200; the player passes the body of the response along with the fault intact.

Using an RPC component with a server-side destination

An RPC component's `destination` property references a Flex Data Services destination configured in the services-config.xml file or a file that it includes by reference. A destination specifies the RPC service class or URL, the transport channel to use, the adapter with which to access the RPC service, and security settings. For more information about configuring destinations, see Chapter 46, "Configuring RPC Services," on page 1451.

To declare a connection to a destination in an MXML tag, you set an `id` property and a `destination` property in one of the three RPC service tags. The `id` property is required for calling the services and handling service results.

The following examples show simple RemoteObject, HTTPService, and WebService component declarations in MXML tags:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCServerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <!-- Declare a RemoteObject component. -->
    <mx:RemoteObject
        id="employeeRO"
        destination="SalaryManager"
        useProxy="true"/>

    <!-- Declare an HTTPService component. -->
    <mx:HTTPService
        id="employeeHTTP"
        destination="SalaryManager"
        useProxy="true"/>

    <!-- Declare a WebService component. -->
    <mx:WebService
        id="employeeWS"
        destination="SalaryManager"
        useProxy="true"/>
</mx:Application>
```

The following examples show simple RemoteObject, HTTPService, and WebService component declarations in ActionScript. You would place the code in these examples inside ActionScript methods:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCNoServerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            // Import required packages.
            import mx.rpc.remoting.RemoteObject;
            import mx.rpc.http.HTTPService;
            import mx.rpc.soap.WebService;
            private var employeeRO:RemoteObject;
            private var employeeHTTP:HTTPService;
            private var employeeWS:WebService;

            public function declareServices():void{
                // Create a RemoteObject component.
                employeeRO = new RemoteObject();
                employeeRO.destination = "SalaryManager";
                // Create an HTTPService component.
                employeeHTTP = new HTTPService();
                employeeHTTP.destination = "SalaryManager";
                // Create a WebService component.
                employeeWS = new WebService();
                employeeWS.destination = "SalaryManager";
            }
        ]]>
    </mx:Script>

</mx:Application>
```

Configuring a destination

You configure a destination in a service definition in the services-config.xml file or a file that it includes by reference. A destination specifies the RPC service class or URL, the transport channel to use, the adapter with which to access the service, and settings for securing the destination.

You configure RemoteObject destinations in the Remoting Service definition in the services-config.xml file or a file that it includes by reference, such as the remoting-config.xml file. You configure HTTP service and web service destinations in the Proxy Service definition in the services-config.xml file or a file that it includes by reference, such as the proxy-config.xml file.

The Proxy Service and its adapters provide functionality that lets applications access HTTP services and web services on different domains. Additionally, the Proxy Service lets you limit access to specific URLs and URL patterns, and provide security. When you do not have Flex Data Services or do not require the functionality provided by the Proxy Service, you can bypass it. You bypass the proxy by setting an HTTPService or WebService component's `useProxy` property to `false`, which is the default value. For more information on the Proxy Service, see Chapter 46, "Configuring RPC Services," on page 1451.

Basic authentication relies on standard J2EE basic authentication from the web application container, and you configure aspects of it in the services-config.xml file and the web.xml file of your Flex web application. You configure custom authentication entirely in services-config.xml file or a file that it includes by reference. For more information about securing service destinations, see Chapter 46, "Configuring RPC Services," on page 1451.

The following example shows a basic server-side configuration for a Remoting Service in the services-config.xml file or a file that it includes by reference, such as the remoting-config.xml file. RemoteObject components connect to Remoting Service destinations. The configuration of HTTP services and web services is very similar to this example.

```
<service id="remoting-service"
  class="flex.messaging.services.RemotingService"
  messageTypes="flex.messaging.messages.RemotingMessage">

  <adapters>
    <adapter-definition id="java-object"
      class="flex.messaging.services.remoting.adapters.JavaAdapter"
      default="true"/>
  </adapters>

  <default-channels>
    <channel ref="samples-amf"/>
    </default-channels>

    <destination id="restaurant">
      <properties>
        <source>samples.restaurant.RestaurantService</source>
        <scope>application</scope>
      </properties>
    </destination>

</service>
```

This Remoting Service destination uses an Action Message Format (AMF) message channel for transporting data. Optionally, it could use one of the other supported message channels. Message channels are defined in the services-config.xml file, in the `<channels>` section under the `services-config` element. For more information about message channels, see Chapter 43, "Configuring message channels," on page 1360.

The Remoting Service destination uses the Java object adapter to connect to a Java object destination, as remote object destinations typically do. For more information about configuring destinations, see Chapter 46, "Configuring RPC Services," on page 1451.

Calling a service

You can declare an RPC component and call the service in MXML or ActionScript. Regardless of the source of input data, calls to a service are asynchronous and require an ActionScript method, called an event listener, which is called after the RPC call is completed.

The two general categories of events are user events and system events. *User events* occur as a result of user interaction with the application; for example, when a user clicks a Button control a user event occurs. *System events* occur as a result of systematic code execution. For information about events, see Chapter 5, "Using Events," on page 83.

Flex provides two ways to call a service: *explicit parameter passing* and *parameter binding*. You can use explicit parameter passing when you declare an RPC component in MXML or ActionScript. Parameter binding is only available when you declare an RPC component in MXML.

You can use an RPC component's `requestTimeout` property to define the number of seconds to allow a request to remain outstanding before timing it out. The `requestTimeout` property is available on RemoteObject, HTTPService, and WebService components.

Using explicit parameter passing

When you use explicit parameter passing, you provide input to a service in the form of parameters to an ActionScript function. This way of calling a service closely resembles the way that you call methods in Java. You cannot use Flex data validators automatically in combination with explicit parameter passing.

Explicit parameter passing with RemoteObject and WebService components

The way you use explicit parameter passing with RemoteObject and WebService components is very similar. The following example shows MXML code for declaring a RemoteObject component and calling a service using explicit parameter passing in the click event listener of a Button control. A ComboBox control provides data to the service. Simple event listeners handle the service-level result and fault events. This example shows `destination` properties, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCParamPassing.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            [Bindable]
            public var empList:Object;
        ]]>
    </mx:Script>

    <mx:RemoteObject
        id="employeeRO"
        destination="SalaryManager"
        result="empList=event.result"
        fault="Alert.show(event.fault.faultString, 'Error');"/>

    <mx:ComboBox id="dept" width="150">
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:source>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:source>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:ComboBox>

    <mx:Button label="Get Employee List"
click="employeeRO.getList(dept.selectedItem.data);"/>
</mx:Application>
```

Explicit parameter passing with HTTPService tags

Explicit parameter passing with HTTPService components is different than it is with RemoteObject and WebService components. You always use an HTTPService component's `send()` method to call a service. This is different from RemoteObject and WebService components, on which you call methods that are client-side versions of the methods or operations of the RPC service.

When you use explicit parameter passing, you can specify an object that contains name-value pairs as a `send()` method parameter. A `send()` method parameter must be a simple base type; you cannot use complex nested objects because there is no generic way to convert them to name-value pairs.

If you do not specify a parameter to the `send()` method, the HTTPService component uses any query parameters specified in an `<mx:request>` tag.

The following examples show two ways to call an HTTP service using the `send()` method with a parameter. The second example also shows how to call the `cancel()` method to cancel an HTTP service call.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCSend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function callService():void {
                // Cancel all previous pending calls.
                myService.cancel();

                var params:Object = new Object();
                params.param1 = 'val1';
                myService.send(params);
            }
        ]]>
    </mx:Script>

    <mx:HTTPService
        id="myService"
        destination="Dest"
        useProxy="true"/>
<!-- HTTP service call with a send() method that takes a variable as its
parameter. The value of the variable is an Object. -->
    <mx:Button click="myService.send({param1: 'val1'});"/>

<!-- HTTP service call with an object as a send() method parameter that
provides query parameters. -->
    <mx:Button click="callService();"/>
</mx:Application>
```

Using parameter binding

Parameter binding lets you copy data from user-interface controls or models to request parameters. Parameter binding is only available for RPC components that you declare in MXML. You can apply validators to parameter values before submitting requests to services. For more information about data binding and data models, see Chapter 38, "Binding Data," on page 1245 and Chapter 39, "Storing Data," on page 1269. For more information about data validation, see Chapter 40, "Validating Data," on page 1281.

When you use parameter binding, you declare RemoteObject method parameter tags nested in an `<mx:arguments>` tag under an `<mx:method>` tag, HTTPService parameter tags nested in an `<mx:request>` tag, or WebService operation parameter tags nested in an `<mx:request>` tag under an `<mx:operation>` tag. You use the `send()` method to send the request.

Parameter binding with RemoteObject components

When you use parameter binding with RemoteObject components, you always declare methods in a RemoteObject component's `<mx:method>` tag.

An `<mx:method>` tag can contain an `<mx:arguments>` tag that contains child tags for the method parameters. The `name` property of an `<mx:method>` tag must match one of the service's method names. The order of the argument tags must match the order of the service's method parameters. You can name argument tags to match the actual names of the corresponding method parameters as closely as possible, but this is not necessary.

NOTE

If argument tags inside an `<mx:arguments>` tag have the same name, service calls fail if the remote method is not expecting an Array as the only input source. There is no warning about this when the application is compiled.

You can bind data to a RemoteObject component's method parameters. You reference the tag names of the parameters for data binding and validation.

The following example shows a method with two parameters bound to the text properties of TextInput controls. A PhoneNumberValidator validator is assigned to `arg1`, which is the name of the first argument tag. This example shows a `destination` property on the RemoteObject component, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\ROParamBind1.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:RemoteObject
        id="ro"
        destination="roDest" >

        <mx:method name="setData">
            <mx:arguments>
                <arg1>{text1.text}</arg1>
                <arg2>{text2.text}</arg2>
            </mx:arguments>
        </mx:method>
    </mx:RemoteObject>
    <mx:TextInput id="text1"/>
    <mx:TextInput id="text2"/>

    <mx:PhoneNumberValidator source="{ro.setData.arguments}"
property="arg1"/>
</mx:Application>
```

Flex sends the argument tag values to the method in the order that the MXML tags specify.

The following example uses parameter binding in a RemoteObject component's `<mx:method>` tag to bind the data of a selected ComboBox item to the `employeeRO.getList` operation when the user clicks a Button control. When you use parameter binding, you call a service by using the `send()` method with no parameters.

```
<?xml version="1.0"?>
<!-- fds\rpc\ROParamBind2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
                import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>
    <mx:RemoteObject
        id="employeeRO"
        destination="roDest"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:method name="getList">
            <mx:arguments>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:arguments>
        </mx:method>
    </mx:RemoteObject>
    <mx:ArrayCollection id="employeeAC"
        source="{ArrayUtil.toArray(employeeRO.getList.lastResult)}"/>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">

            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
            <mx:Button label="Get Employee List"
                click="employeeRO.getList.send()"/>
    </mx:HBox>
    <mx:DataGrid dataProvider="{employeeAC}" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
```

```
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

If you are unsure whether the result of a service call contains an Array or an individual object, you can use the `toArray()` method of the mx.utils.ArrayUtil class to convert it to an Array, as this example shows. If you pass the `toArray()` method to an individual object, it returns an Array with that object as the only Array element. If you pass an Array to the method, it returns the same Array. For information about working with ArrayCollection objects, see Chapter 7, "Using Data Providers and Collections," on page 161.

Parameter binding with HTTPService components

When an HTTP service takes query parameters, you can declare them as child tags of an `<mx:request>` tag. The names of the tags must match the names of the query parameters that the service expects.

The following example uses parameter binding in an HTTPService component's `<mx:request>` tag to bind the data of a selected ComboBox item to the `employeeSrv` request when the user clicks a Button control. When you use parameter binding, you call a service by using the `send()` method with no parameters. This example shows a `url` property on the HTTPService component, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\HttpServiceParamBind.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="20">
    <mx:Script>
        <![CDATA[
            import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>

    <mx:HTTPService
        id="employeeSrv"
        url="employees.jsp">
        <mx:request>
            <deptId>{dept.selectedItem.data}</deptId>
        </mx:request>
    </mx:HTTPService>
    <mx:ArrayCollection
        id="employeeAC"
        source=

"{ArrayUtil.toArray(employeeSrv.lastResult.employees.employee)}"/>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeSrv.send();"/>
    </mx:HBox>
```

```
    <mx:DataGrid dataProvider="{employeeAC}"
        width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

If you are unsure whether the result of a service call contains an Array or an individual object, you can use the `toArray()` method of the mx.utils.ArrayUtil class to convert it to an Array, as the previous example shows. If you pass the `toArray()` method to an individual object, it returns an Array with that object as the only Array element. If you pass an Array to the method, it returns the same Array. For information about working with ArrayCollection objects, see Chapter 7, "Using Data Providers and Collections," on page 161.

Parameter binding with WebService components

When you use parameter binding with a WebService component, you always declare operations in the WebService component's `<mx:operation>` tags. An `<mx:operation>` tag can contain an `<mx:request>` tag that contains the XML nodes that the operation expects. The `name` property of an `<mx:operation>` tag must match one of the web service operation names.

You can bind data to parameters of web service operations. You reference the tag names of the parameters for data binding and validation.

The following example uses parameter binding in a WebService component's `<mx:operation>` tag to bind the data of a selected ComboBox item to the `employeeWS.getList` operation when the user clicks a Button control. The `<deptId>` tag corresponds directly to the `getList` operation's `deptId` parameter. When you use parameter binding, you call a service by using the `send()` method with no parameters. This example shows a `destination` property on the WebService component, but the way you call a service is the same whether you connect to the service directly or go through a destination.

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceParamBind.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.utils.ArrayUtil;
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:WebService
        id="employeeWS"
        destination="wsDest"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString)">
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:ArrayCollection
        id="employeeAC"
        source="{ArrayUtil.toArray(employeeWS.getList.lastResult)}"/>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
            <mx:Button label="Get Employee List"
                click="employeeWS.getList.send()"/>
    </mx:HBox>
    <mx:DataGrid dataProvider="{employeeAC}" width="100%">
```

```
        <mx:columns>
                <mx:DataGridColumn dataField="name" headerText="Name"/>
                <mx:DataGridColumn dataField="phone" headerText="Phone"/>
              <mx:DataGridColumn dataField=" to email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

You can also manually specify an entire SOAP request body in XML with all of the correct namespace information defined in an `<mx:request>` tag. To do this, you must set the value of the `format` attribute of the `<mx:request>` tag to `xml`, as the following example shows:

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceSOAPRequest.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:WebService id="ws" wsdl="http://api.google.com/GoogleSearch.wsdl"
        useProxy="true">
        <mx:operation name="doGoogleSearch" resultFormat="xml">
            <mx:request format="xml">
                <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                    <key xsi:type="xsd:string">XYZ123</key>
                    <q xsi:type="xsd:string">Balloons</q>
                    <start xsi:type="xsd:int">0</start>
                    <maxResults xsi:type="xsd:int">10</maxResults>
                    <filter xsi:type="xsd:boolean">true</filter>
                    <restrict xsi:type="xsd:string"/>
                    <safeSearch xsi:type="xsd:boolean">false</safeSearch>
                    <lr xsi:type="xsd:string" />
                    <ie xsi:type="xsd:string">latin1</ie>
                    <oe xsi:type="xsd:string">latin1</oe>
                </ns1:doGoogleSearch>
            </mx:request>
        </mx:operation>
    </mx:WebService>
</mx:Application>
```

Setting properties for RemoteObject methods or WebService operations

You can set the following properties on a RemoteObject component's `<mx:method>` tag or a WebService component's `<mx:operation>` tag. The `name` property is the only property that is required. The following table describes the properties:

Property	Description
`concurrency`	Value that indicates how to handle multiple calls to the same method. By default, making a new request to an operation or method that is already executing does not cancel the existing request. The following values are permitted: • `multiple` Existing requests are not cancelled, and the developer is responsible for ensuring the consistency of returned data by carefully managing the event stream. This is the default value. • `single` Making only one request at a time is allowed on the method; multiple requests generate a fault. • `last` Making a request cancels any existing request. **Note:** The request referred to here is not the HTTP request. It is the client action request, or the pendingCall object. HTTP requests are sent to the server and get processed on the server side. However, the result is ignored when a request is cancelled (requests are cancelled when you use the `single` or `last` value). The `last` request is not necessarily the last one that the server receives over HTTP.
`fault`	ActionScript code that runs when an error occurs. The fault is passed as an event parameter.
`name`	(Required) Name of the operation or method to call.
`result`	ActionScript code that runs when a lastResult object is available. The result object is passed as an event parameter.

Calling services in ActionScript

You always use explicit parameter passing when you call a service in ActionScript. This section shows ActionScript examples for RemoteObject, HTTPService, and WebService components.

Calling RemoteObject components in ActionScript

The following ActionScript example is equivalent to the MXML example in "Explicit parameter passing with RemoteObject and WebService components" on page 1414. Calling the `useRemoteObject()` method declares the service, sets the destination, sets up result and fault event listeners, and calls the service's `getList()` method.

```
<?xml version="1.0"?>
<!-- fds\rpc\ROInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.remoting.RemoteObject;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            [Bindable]
            public var empList:Object;
            public var employeeRO:RemoteObject;

           public function useRemoteObject(intArg:int, strArg:String):void
{
                employeeRO = new RemoteObject();
                employeeRO.destination = "SalaryManager";
                employeeRO.getList.addEventListener("result",
getListResultHandler);
                employeeRO.addEventListener("fault", faultHandler);
                employeeRO.getList(deptComboBox.selectedItem.data);
            }

            public function getListResultHandler(event:ResultEvent):void {
                 // Do something
                empList=event.result;
            }

            public function faultHandler (event:FaultEvent):void {
             // Deal with event.fault.faultString, etc.
                Alert.show(event.fault.faultString, 'Error');
            }
        ]]>
    </mx:Script>
    <mx:ComboBox id="deptComboBox"/>
</mx:Application>
```

Calling web services in ActionScript

The following example shows a web service call in an ActionScript script block. Calling the `useWebService()` method declares the service, sets the destination, fetches the WSDL document, and calls the service's `echoArgs()` method. Notice that you must call the `WebService.loadWSDL()` method when you declare a WebService component in ActionScript.

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
        import mx.rpc.soap.WebService;
        import mx.rpc.events.ResultEvent;
        import mx.rpc.events.FaultEvent;
        private var ws:WebService;
        public function useWebService(intArg:int, strArg:String):void {
            ws = new mx.rpc.soap.WebService();
            ws.destination = "echoArgService";
            ws.echoArgs.addEventListener("result", echoResultHandler);
            ws.addEventListener("fault", faultHandler);
            ws.loadWSDL();
            ws.echoArgs(intArg, strArg);
        }

        public function echoResultHandler(event:ResultEvent):void {
            var retStr:String = event.result.echoStr;
            var retInt:int = event.result.echoInt;
         //Do something.
        }

        public function faultHandler(event:FaultEvent):void {
      //deal with event.fault.faultString, etc
        }
        ]]>
    </mx:Script>
</mx:Application>
```

Calling HTTP services in ActionScript

The following example shows an HTTP service call in an ActionScript script block. Calling the `useHTTPService()` method declares the service, sets the destination, sets up result and fault event listeners, and calls the service's `send()` method.

```
<?xml version="1.0"?>
<!-- fds\rpc\HttpServiceInAS.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.http.HTTPService;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            private var service:HTTPService

            public function useHttpService(parameters:Object):void {
                service = new HTTPService();
                service.destination = "sampleDestination";
                service.method = "POST";
                service.addEventListener("result", httpResult);
                service.addEventListener("fault", httpFault);
                service.send(parameters);
            }

            public function httpResult(event:ResultEvent):void {
                var result:Object = event.result;
            //Do something with the result.
            }

            public function httpFault(event:FaultEvent):void {
                var faultstring:String = event.fault.faultString;
                Alert.show(faultstring);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Handling service results

After an RPC component calls a service, the data that the service returns is placed in a lastResult object. By default, the `resultFormat` property value of HTTPService components and WebService component operations is `object`, and the data that is returned is represented as a simple tree of ActionScript objects. Flex interprets the XML data that a web service or HTTP service returns to appropriately represent base types, such as String, Number, Boolean, and Date. To work with strongly typed objects, you must populate those objects using the object tree that Flex creates.

WebService and HTTPService components both return anonymous Objects and Arrays that are complex types. If `makeObjectsBindable` is `true`, which it is by default, Objects are wrapped in mx.utils.ObjectProxy instances and Arrays are wrapped in mx.collections.ArrayCollection instances.

NOTE	Macromedia ColdFusion is case insensitive, so it internally uppercases all of its data. Keep this in mind when consuming a ColdFusion web service.

Handling results as XML with the e4x result format

You can set the `resultFormat` property value of HTTPService components and WebService operations to `e4x` to create a lastResult object of type XML. You can access the lastResult object by using ECMAScript for XML (E4X) expressions. Using a `resultFormat` of `e4x` is the preferred way to work with XML, but you can also set the `resultFormat` property to `xml` to create a lastResult object of type flash.xml.XMLNode, which is a legacy object for working with XML. Also, you can set the `resultFormat` property of HTTPService components to `flashvars` or `text` to create results as ActionScript objects that contain name-value pairs or as raw text, respectively. For more information, see *Adobe Flex 2 Language Reference*.

NOTE	If you want to use E4X syntax on service results, you must set the `resultFormat` property of your HTTPService or WebService component to `e4x`. The default value is `object`.

When you set the `resultFormat` property of a WebService operation to `e4x`, you may have to handle namespace information contained in the body of the SOAP envelope that the web service returns. The following example shows part of a SOAP body that contains namespace information. This data was returned by a web service that gets stock quotes. The namespace information is in boldface text.

```
...
<soap:Body>
<GetQuoteResponse
xmlns="http://ws.invesbot.com/">
<GetQuoteResult><StockQuote xmlns="">
<Symbol>ADBE</Symbol>
<Company>ADOBE SYSTEMS INC</Company>
<Price>&lt;big&gt;&lt;b&gt;35.90&lt;/b&gt;&lt;/big&gt;</Price>
...
</soap:Body>
...
```

Because this soap:Body contains namespace information, if you set the `resultFormat` property of the WebService operation to `e4x`, you must create a namespace object for the http://ws.invesbot.com/ namespace. The following example shows an application that does that:

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceE4XResult1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
    pageTitle="Test" >
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            private namespace invesbot = "http://ws.invesbot.com/";
            use namespace invesbot;
        ]]>
    </mx:Script>
    <mx:WebService
        id="WS"
        destination="stockservice" useProxy="true"
        fault="Alert.show(event.fault.faultString), 'Error'">
        <mx:operation name="GetQuote" resultFormat="e4x">
            <mx:request>
                <symbol>ADBE</symbol>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:HBox>
        <mx:Button label="Get Quote" click="WS.GetQuote.send()"/>
        <mx:Text
            text="{WS.GetQuote.lastResult.GetQuoteResult.StockQuote.Price}"
        />
    </mx:HBox>
</mx:Application>
```

Optionally, you can create a var for a namespace and access it in a binding to the service result, as the following example shows:

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceE4XResult2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
    pageTitle="Test" >
    <mx:Script>
        <![CDATA[
        import mx.controls.Alert;
        public var invesbot:Namespace =
            new Namespace("http://ws.invesbot.com/");
        ]]>
    </mx:Script>
    <mx:WebService
        id="WS"
        destination="stockservice" useProxy="true"
        fault="Alert.show(event.fault.faultString), 'Error'">
        <mx:operation name="GetQuote" resultFormat="e4x">
            <mx:request>
                <symbol>ADBE</symbol>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:HBox>
        <mx:Button label="Get Quote" click="WS.GetQuote.send()"/>
        <mx:Text

text="{WS.GetQuote.lastResult.invesbot::GetQuoteResult.StockQuote.Price}"
        />
    </mx:HBox>
</mx:Application>
```

You use E4X syntax to access elements and attributes of the XML that is returned in a lastResult object. You use different syntax, depending on whether there is a namespace or namespaces declared in the XML.

No namespace

The following example shows how to get an element or attribute value when no namespace is specified on the element or attribute:

```
var attributes:XMLList = XML(event.result).Description.value;
```

The previous code returns `xxx` for the following XML document:

```
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <Description>
    <value>xxx</value>
  </Description>
</RDF>
```

Any namespace

The following example shows how to get an element or attribute value when any namespace is specified on the element or attribute:

```
var attributes:XMLList = XML(event.result).*::Description.*::value;
```

The previous code returns `xxx` for either one of the following XML documents:

XML document one:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description>
    <rdf:value>xxx</rdf:xxx>
  </rdf:Description>
</rdf:RDF>
```

XML document two:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:cm="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <cm:Description>
    <rdf:value>xxx</rdf:xxx>
  </cm:Description>
</rdf:RDF>
```

Specific namespace

The following example shows how to get an element or attribute value when the declared rdf namespace is specified on the element or attribute:

```
var rdf:Namespace = new Namespace("http://www.w3.org/1999/02/22-rdf-syntax-
  ns#");
var attributes:XMLList = XML(event.result).rdf::Description.rdf::value;
```

The previous code returns `xxx` for the following XML document:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description>
    <rdf:value>xxx</rdf:xxx>
  </rdf:Description>
</rdf:RDF>
```

The following example shows an alternate way to get an element or attribute value when the declared rdf namespace is specified on an element or attribute:

```
namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
use namespace rdf;
var attributes:XMLList = XML(event.result).rdf::Description.rdf::value;
```

The previous code also returns `xxx` for the following XML document:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description>
    <rdf:value>xxx</rdf:xxx>
  </rdf:Description>
</rdf:RDF>
```

Binding a service result to other objects

You can bind properties of an RPC component's lastResult object to the properties of other objects, including user-interface components and data models. The lastResult is the result object from the last invocation. Whenever a service request executes, the lastResult is updated and any associated bindings are also updated.

In the following example, two properties of the lastResult object, `CityShortName` and `CurrentTemp`, are bound to the `text` properties of two TextArea controls. The `CityShortName` and `CurrentTemp` properties are returned when a user makes a request to the `MyService.GetWeather()` operation and provides a ZIP code as an operation request parameter.

```
<mx:TextArea text="{MyService.GetWeather.lastResult.CityShortName}"/>
<mx:TextArea text="{MyService.GetWeather.lastResult.CurrentTemp}"/>
```

You can bind a lastResult object to the `source` property of an ArrayCollection object. When you use an HTTPService or WebService component, you can bind a lastResult to an XMLListCollection object when the HTTPService component's or WebService operation's `resultFormat` property is set to `e4x`. There is no `resultFormat` property on RemoteObject methods. You can then bind the ArrayCollection object or XMLListCollection object to a complex property of a user-interface component, such as a List, ComboBox, or DataGrid control.

You can use the ArrayCollection or XMLListCollection API to work with the data. When using an XMLListCollection object, you can use ECMAScript for XML (E4X) expressions to work with the data.

Binding a result to an ArrayCollection object

In the following example, a service lastResult object, `employeeWS.getList.lastResult`, is bound to the `source` property of an ArrayCollection object. The ArrayCollection object is bound to the `dataProvider` property of a DataGrid control that displays employees' names, phone numbers, and e-mail addresses.

```
<?xml version="1.0"?>
<!-- fds\rpc\BindingResultArrayCollection.mxml. Warnings on mx:Object  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>
    <mx:WebService id="employeeWS" destination="employeeWS"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:ArrayCollection id="ac"
        source="{ArrayUtil.toArray(employeeWS.getList.lastResult)}"/>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List"
click="employeeWS.getList.send()"/>
    </mx:HBox>
    <mx:DataGrid dataProvider="{ac}" width="100%">
        <mx:columns>
```

```
                <mx:DataGridColumn dataField="name" headerText="Name"/>
                <mx:DataGridColumn dataField="phone" headerText="Phone"/>
                <mx:DataGridColumn dataField="email" headerText="Email"/>
            </mx:columns>
        </mx:DataGrid>
</mx:Application>
```

If you are unsure whether the result of a service call contains an Array or an individual object, you can use the `toArray()` method of the mx.utils.ArrayUtil class to convert it to an Array, as the previous example shows. If you pass the `toArray()` method to an individual object, it returns an Array with that object as the only Array element. If you pass an Array to the method, it returns the same Array.

For information about working with ArrayCollection objects, see Chapter 7, "Using Data Providers and Collections," on page 161.

Binding a result to an XMLListCollection object

In the following example, a service lastResult object, `employeeWS.getList.lastResult`, is bound to the `source` property of an XMLListCollection object. The XMLListCollection object is bound to the `dataProvider` property of a DataGrid control that displays employees' names, phone numbers, and e-mail addresses.

NOTE To bind service results to an XMLListCollection, you must set the `resultFormat` property of your HTTPService or WebService component to `e4x`. The default value of this property is `object`.

```
<?xml version="1.0"?>
<!-- fds\rpc\BindResultXMLListCollection.mxml. Warnings on mx:Object -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </mx:Script>
    <mx:WebService id="employeeWS"
        destination="employeeWS"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:operation name="getList" resultFormat="e4x">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
```

```
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List"
            click="employeeWS.getList.send()"/>
    </mx:HBox>
    <mx:XMLListCollection id="xc"
        source="{employeeWS.getList.lastResult}"/>
    <mx:DataGrid dataProvider="{xc}" width="100%">
        <mx:columns>
                <mx:DataGridColumn dataField="name" headerText="Name"/>
                <mx:DataGridColumn dataField="phone" headerText="Phone"/>
                <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

For information about working with XMLListCollection objects, see Chapter 7, “Using Data Providers and Collections,” on page 161.

Handling result and fault events

When a service call is completed, the RemoteObject method, WebService operation, or HTTPService component dispatches a result event or a fault event. A *result event* indicates that the result is available. A *fault event* indicates that an error occurred. The result event acts as a trigger to update properties that are bound to the lastResult. You can handle fault and result events explicitly by adding event listeners to RemoteObject methods or WebService operations. For an HTTPService component, you specify result and fault event listeners on the component itself because an HTTPService component does not have multiple operations or methods.

When you do not specify event listeners for result or fault events on a RemoteObject method or a WebService operation, the events are passed to the component level; you can specify component-level result and fault event listeners.

In the following MXML example, the `result` and `fault` events of a WebService operation specify event listeners; the `fault` event of the WebService component also specifies an event listener:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCResultFaultMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;
            public function showErrorDialog(event:FaultEvent):void {
                // Handle operation fault.
                Alert.show(event.fault.faultString, "Error");
            }
            public function defaultFault(event:FaultEvent):void {
                // Handle service fault.
               Alert.show(event.fault.faultString, "Error");
            }
            public function log(event:ResultEvent):void {
                // Handle result.
            }
        ]]>
    </mx:Script>
    <mx:WebService id="WeatherService" destination="wsDest"
        fault="defaultFault(event)">
        <mx:operation name="GetWeather"
            fault="showErrorDialog(event)"
            result="log(event)">
            <mx:request>
                <ZipCode>{myZip.text}</ZipCode>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:TextInput id="myZip"/>
</mx:Application>
```

In the following ActionScript example, a result event listener is added to a WebService operation; a fault event listener is added to the WebService component:

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCResultFaultAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.WebService;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            private var ws:WebService;

            public function useWebService(intArg:int, strArg:String):void {
                ws = new WebService();
                ws.destination = "wsDest";
                ws.echoArgs.addEventListener("result", echoResultHandler);
                ws.addEventListener("fault", faultHandler);
                ws.loadWSDL();
                ws.echoArgs(intArg, strArg);
            }

            public function echoResultHandler(event:ResultEvent):void {
                var retStr:String = event.result.echoStr;
                var retInt:int = event.result.echoInt;
                //do something
        }

            public function faultHandler(event:FaultEvent):void {
            //deal with event.fault.faultString, etc.
            }
        ]]>
    </mx:Script>
</mx:Application>
```

You can also use the mx.rpc.events.InvokeEvent event to to indicate when an RPC component request has been invoked. This is useful if operations are queued and invoked at a later time.

Using a service with binding, validation, and event listeners

You can validate data before passing it to a service, and dispatch an event when the service returns a result or a fault. The following example shows an application that validates service request data and assigns an event listener to result and fault events.

This two-tier application does the following:

1. Declares a web service.
2. Binds user-interface data to a web service request.
3. Validates a ZIP code.
4. Binds data from a web service result to a user-interface control.
5. Specifies result and fault event listeners for a WebService operation.

You can also create multitier applications that use an additional data-model layer between the user interface and the web service. For more information about data models and data binding, see Chapter 37, “Representing Data,” on page 1237. For more information about data validation, see Chapter 40, “Validating Data,” on page 1281.

```
<?xml version="1.0"?>
<!-- fds\rpc\RPCBindValidateEvents.mxml. Compiles w destination error -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600" height="400">

    <!-- WebService component handles web service requests and results. -->
    <mx:WebService id="WeatherService" destination="wsDest">
        <mx:operation name="GetWeather"
            fault="showErrorDialog(event.fault.faultString)"
result="log();">

    <!-- The mx:request data model stores web service request data. -->
            <mx:request>
                <ZipCode>{myZipField.text}</ZipCode>
            </mx:request>
        </mx:operation>
    </mx:WebService>

   <!-- Validator validates ZIP code using the standard Zip Code validator.
-->
    <mx:ZipCodeValidator
        source="{WeatherService.GetWeather.request}" property="ZipCode"
        trigger="{mybutton}" triggerEvent="click"/>
    <mx:VBox>
    <mx:TextInput id="myZipField" text="enter zip" width="80"/>
```

```
    <!-- Button triggers service request. -->
    <mx:Button id="mybutton" label="Get Weather"
        click="WeatherService.GetWeather.send();"/>

   <!-- TextArea control displays the results that the service returns. -->
        <mx:TextArea id="temp" text="The current temperature in
        {WeatherService.GetWeather.lastResult.CityShortName} is
        {WeatherService.GetWeather.lastResult.CurrentTemp}."
        height="30" width="200"/>
    </mx:VBox>
    <mx:Script>
        <![CDATA[
            public function log():void {
                // function implementation
            }

            public function showErrorDialog(error:String):void {
                // function implementation
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Handling asynchronous calls to services

Because ActionScript code executes asynchronously, if you allow concurrent calls to a service, you must ensure that your code handles the results appropriately, based on the context in which the service is called. By default, making a request to a web service operation that is already executing does not cancel the existing request. In a Flex application in which a service can be called from multiple locations, the service might respond differently in different contexts.

When you design a Flex application, consider whether the application requires disparate data sources, and the number of types of services that the application requires. The answers to these questions help determine the level of abstraction that you provide in the data layer of the application.

In a very simple application, user-interface components might call services directly. In applications that are slightly larger, business objects might call services. In still larger applications, business objects might interact with service broker objects that call services.

To understand the results of asynchronous service calls to objects in an application, you need a good understanding of scoping in ActionScript.

Using the Asynchronous Completion Token design pattern

Flex is a service-oriented framework in which code executes asynchronously, therefore, it lends itself well to the Asynchronous Completion Token (ACT) design pattern. This design pattern efficiently dispatches processing within a client in response to the completion of asynchronous operations that the client invokes. For more information, see www.cs.wustl.edu/~schmidt/PDF/ACT.pdf.

When you use the ACT design pattern, you associate application-specific actions and state with responses that indicate the completion of asynchronous operations. For each asynchronous operation, you create an ACT that identifies the actions and state that are required to process the results of the operation. When the result is returned, you can use its ACT to distinguish it from the results of other asynchronous operations. The client uses the ACT to identify the state required to handle the result.

An ACT for a particular asynchronous operation is created before the operation is called. While the operation is executing, the client continues executing. When the service sends a response, the client uses the ACT that is associated with the response to perform the appropriate actions.

When you call a Flex remote object service, web service, or HTTP service, Flex returns an instance of the service call. When you use the default `concurrency` value, `multiple`, you can use the call object that is returned by the data service's `send()` method to handle the specific results of each concurrent call to the same service. You can add information to this call object when it is returned, and then in a result event listener you can pass back the call object as event.token. This is an implementation of the ACT design pattern that uses the call object of each data service call as an ACT. How you use the ACT design pattern in your own code depends on your requirements. For example, you might attach simple identifiers to individual calls, more complex objects that perform their own set of functionality, or functions that a central listener calls.

The following example shows a simple implementation of the ACT design pattern. This example uses an HTTP service and attaches a simple variable to the call object.

```
...
<mx:HTTPService id="MyService" destination="httpDest"
  result="resultHandler(event)"/>
...
<mx:Script>
  <![CDATA[
    ...
    public function storeCall():void {
      // Create a variable called call to store the instance
      // of the service call that is returned.
```

```
        var call:Object = MyService.send();

        // Add a variable to the call object that is returned.
        // You can name this variable whatever you want.
        call.marker = "option1";
        ...
      }

      // In a result event listener, execute conditional
      // logic based on the value of call.marker.
      private function resultHandler(event:ResultEvent):void {
        var call:object = event.token
        if (call.marker == "option1") {
        //do option 1
        }
        else
        ...
      }
    ]]>
</mx:Script>
...
```

Making a service call when another call is completed

Another common requirement when using data services is the dependency of one service call on the result of another. Your code must not make the second call until the result of the first call is available. You must make the second service call in the result event listener of the first, as the following example shows:

```
...
<mx:WebService id="ws" destination="wsDest"...>
  <mx:operation name="getCurrentSales"
  result="resultHandler(event.result)"/>
  <mx:operation name="setForecastWithSalesInput"/>
</mx:WebService>
<mx:Script>
  <![CDATA[
    // Call the getForecastWithSalesInput operation with the result of the
    // getCurrentSales operation.
      public function resultHandler(currentsales:String):void {
      ws.setForecastWithSalesInput(currentsales);
      //Or some variation that uses data binding.
      }
  ]]>
</mx:Script>
...
```

Using features specific to RemoteObject components

You can use a RemoteObject component to call methods on a Remoting Services destination, which specifies a Java object that resides on the application server on which Flex Data Services is running and is in the web application's source path.

Accessing Java objects in the source path

The RemoteObject component lets you access stateless and stateful objects that are in the Flex Data Services web application's source path. You can place stand-alone class files in the web application's WEB-INF/classes directory to add them to the source path. You can place classes contained in Java Archive (JAR) files in the web application's WEB-INF/lib directory to add them to the source path. You must specify the fully qualified class name in the `source` property of a Remoting Service destination in the services-config.xml file or a file that it includes by reference, such as the remoting-config.xml file. The class also must have a no-args constructor. For information about configuring Remoting Service destinations, see Chapter 46, "Configuring RPC Services".

When you configure a Remoting Service destination to access stateless objects (the request scope), Flex creates a new object for each method call instead of calling methods on the same object. You can set the scope of an object to the request scope (default value), the application scope, or the session scope. Objects in the application scope are available to the web application that contains the object. Objects in the session scope are available to the entire client session.

When you configure remote object destination to access stateful objects, Flex creates the object once on the server and maintains state between method calls. You should use the request scope if storing the object in the application or session scope causes memory problems.

Accessing EJBs and other objects in JNDI

You can access Enterprise JavaBeans (EJBs) and other objects stored in the Java Naming and Directory Interface (JNDI) by calling methods on a destination that is a service facade class that looks up an object in JNDI and calls its methods.

You can use stateless or stateful objects to call the methods of Enterprise JavaBeans and other objects that use JNDI. For an EJB, you can call a service facade class that returns the EJB object from JNDI and calls a method on the EJB.

In your Java class, you use the standard Java coding pattern, in which you create an initial context and perform a JNDI lookup. For an EJB, you also use the standard coding pattern in which your class contains methods that call the EJB home object's `create()` method and the resulting EJB's business methods.

The following example uses a method called `getHelloData()` on a facade class destination:

```
<mx:RemoteObject id="Hello" destination="roDest">
   <mx:method name="getHelloData"/>
</mx:RemoteObject>
```

On the Java side, the `getHelloData()` method could easily encapsulate everything necessary to call a business method on an EJB. The Java method in the following example performs the following actions:

- Creates new initial context for calling the EJB
- Performs a JNDI lookup that gets an EJB home object
- Calls the EJB home object's `create()` method
- Calls the EJB's `sayHello()` method

```
...
public void getHelloData() {
  try{
    InitialContext ctx = new InitialContext();

    Object obj = ctx.lookup("/Hello");

    HelloHome ejbHome = (HelloHome)

    PortableRemoteObject.narrow(obj, HelloHome.class);

    HelloObject ejbObject = ejbHome.create();

    String message = ejbObject.sayHello();
    }
  catch (Exception e);
  }
...
```

Reserved method names

The Flex remoting library uses the following method names; do not use these as your own method names:

```
addHeader()
addProperty()
deleteHeader()
hasOwnProperty()
isPropertyEnumerable()
isPrototypeOf()
registerClass()
toLocaleString()
toString()
unwatch()
valueOf()
watch()
```

You also should not begin method names with an underscore (_) character.

RemoteObject and WebService Operations (methods) are usually accessible by simply naming them after the service variable. However, if your Operation name happens to match a defined method on the service, you can use the following method in ActionScript on a RemoteObject or WebService component to return the Operation of the given name:

```
public function getOperation(name:String):Operation
```

Using features specific to WebService components

Flex applications can interact with web services that define their interfaces in a Web Services Description Language 1.1 (WSDL 1.1) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages. Flex supports RPC-encoded and document-literal web services. Web services use either RPC- and document-style SOAP bindings. The two most common types of web services use RPC-encoded or document-literal SOAP bindings; the terms *encoded* and *literal* indicate the type of WSDL-to-SOAP mapping that a service uses.

NOTE	Flex does not support the following XML schema types: choice, union, default, list, or group. Flex also does not support the following data types: duration, gMonth, gYear, gYearMonth, gDay, gMonthDay, Name, Qname, NCName, anyURI, or language. (Flex supports anyURL but treats it like a String.

Flex applications support web service requests and results that are formatted as Simple Object Access Protocol (SOAP) messages. SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

Adobe Flash Player operates within a security sandbox that limits what Flex applications and other Flash applications can access over HTTP. Flash applications are only allowed HTTP access to resources on the same domain and by the same protocol from which they were served. This presents a problem for web services, because they are typically accessed from remote locations. The Flex proxy, available in Flex Data Services, intercepts requests to remote web services, redirects the requests, and then returns the responses to the client.

If you are not using Flex Data Services, you can access web services in the same domain as your Flex application or a crossdomain.xml (cross-domain policy) file that allows access from your application's domain must be installed on the web server hosting the RPC service. For more information about crossdomain.xml files, see Chapter 4, "Applying Flex Security" in *Building and Deploying Flex 2 Applications*.

Reading WSDL documents

You can view a WSDL document in a web browser, a simple text editor, an XML editor, or a development environment such as Adobe Dreamweaver, which contains a built-in utility for displaying WSDL documents in an easy-to-read format.

A WSDL document contains the tags described in the following table:

Tag	Description
`<binding>`	Specifies the protocol that clients, such as Flex applications, use to communicate with a web service. Bindings exist for SOAP, HTTP GET, HTTP POST, and MIME. Flex supports the SOAP binding only.
`<fault>`	Specifies an error value that is returned as a result of a problem processing a message.
`<input>`	Specifies a message that a client, such as a Flex application, sends to a web service.
`<message>`	Defines the data that a web service operation transfers.
`<operation>`	Defines a combination of `<input>`, `<output>`, and `<fault>` tags.
`<output>`	Specifies a message that the web service sends to a web service client, such as a Flex application.
`<port>`	Specifies a web service endpoint, which specifies an association between a binding and a network address.

Tag	Description
<portType>	Defines one or more operations that a web service provides.
<service>	Defines a collection of <port> tags. Each service maps to one <portType> tag and specifies different ways to access the operations in that <portType> tag.
<types>	Defines data types that a web service's messages use.

RPC-oriented operations and document-oriented operations

A WSDL file can specify either remote procedure call (RPC) oriented or document-oriented (document/literal) operations. Flex supports both operation styles.

When calling an RPC-oriented operation, a Flex application sends a SOAP message that specifies an operation and its parameters. When calling a document-oriented operation, a Flex application sends a SOAP message that contains an XML document.

In a WSDL document, each <port> tag has a binding property that specifies the name of a particular <soap:binding> tag, as the following example shows:

```
<binding name="InstantMessageAlertSoap" type="s0:InstantMessageAlertSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document"/>
```

The style property of the associated <soap:binding> tag determines the operation style. In this example, the style is document.

Any operation in a service can specify the same style or override the style that is specified for the port associated with the service, as the following example shows:

```
<operation name="SendMSN">
  <soap:operation soapAction="http://www.bindingpoint.com/ws/imalert/
    SendMSN"style="document"/>
```

Stateful web services

Flex uses Java server sessions to maintain the state of web service endpoints that use cookies to store session information. This feature acts as an intermediary between Flex applications and web services. It adds an endpoint's identity to whatever the endpoint passes to a Flex application. If the endpoint sends session information, the Flex application receives it. This feature requires no configuration; it is not supported for destinations that use the RTMP channel when using the Proxy Service.

Working with SOAP headers

A SOAP header is an optional tag in a SOAP envelope that usually contains application-specific information, such as authentication information. This section describes how to add SOAP headers to web service requests, clear SOAP headers, and get SOAP headers that are contained in web service results.

Adding SOAP headers to web service requests

Some web services require that you pass along a SOAP header when you call an operation.

You can add a SOAP header to all web service operations or individual operations by calling a WebService or Operation object's `addHeader()` method or `addSimpleHeader()` method in an event listener function.

When you use the `addHeader()` method, you first must create SOAPHeader and QName objects separately. The `addHeader()` method has the following signature:

```
addHeader(header:mx.rpc.soap.SOAPHeader):void
```

To create a SOAPHeader object, you use the following constructor:

```
SOAPHeader(qname:QName, content:Object)
```

To create the QName object in the first parameter of the `SOAPHeader()` method, you use the following constructor:

```
QName(uri:String, localName:String)
```

The `content` parameter of the `SOAPHeader()` constructor is a set of name-value pairs based on the following format:

```
{name1:value1, name2:value2}
```

The `addSimpleHeader()` method is a shortcut for a single name-value SOAP header. When you use the `addSimpleHeader()` method, you create SOAPHeader and QName objects in parameters of the method. The `addSimpleHeader()` method has the following signature:

```
addSimpleHeader(qnameLocal:String, qnameNamespace:String,
  headerName:String, headerValue:Object):void
```

The `addSimpleHeader()` method takes the following parameters:

- `qnameLocal` is the local name for the header QName.
- `qnameNamespace` is the namespace for the header QName.
- `headerName` is the name of the header.
- `headerValue` is the value of the header. This can be a String if it is a simple value, an Object that will undergo basic XML encoding, or XML if you want to specify the header XML yourself.

The code in the following example shows how to use the `addHeader()` method and the `addSimpleHeader()` method to add a SOAP header. The methods are called in an event listener function called `headers`, and the event listener is assigned in the `load` property of an `<mx:WebService>` tag:

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceAddHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="600">
    <mx:WebService id="ws" destination="wsDest" load="headers();"/>
    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.SOAPHeader;
            private var header1:SOAPHeader;
            private var header2:SOAPHeader;

            public function headers():void {

            // Create QName and SOAPHeader objects.
           var q1:QName=new QName("http://soapinterop.org/xsd", "Header1");
            header1=new SOAPHeader(q1, {string:"bologna",int:"123"});
            header2=new SOAPHeader(q1, {string:"salami",int:"321"});


            // Add the header1 SOAP Header to all web service requests.
            ws.addHeader(header1);

            // Add the header2 SOAP Header to the getSomething operation.
            ws.getSomething.addHeader(header2);

            // Within the addSimpleHeader method,
            // which adds a SOAP header to web
            //service requests, create SOAPHeader and QName objects.
            ws.addSimpleHeader
                ("header3", "http://soapinterop.org/xsd", "foo","bar");
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Clearing SOAP headers

You use a WebService or Operation object's `clearHeaders()` method to remove SOAP headers that you added to the object, as the following example shows for a WebService object. You must call `clearHeaders()` at the level (WebService or Operation) where the header was added.

```
<?xml version="1.0"?>
<!-- fds\rpc\WebServiceClearHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="600" >

    <!-- The value of the destination property is for demonstration only and
is not a real destination. -->

    <mx:WebService id="ws" destination="wsDest" load="headers();"/>

    <mx:Script>
        <![CDATA[
            import mx.rpc.*;
            import mx.rpc.soap.SOAPHeader;

            private function headers():void {
                // Create QName and SOAPHeader objects.
                var q1:QName=new QName("Header1", "http://soapinterop.org/
xsd");
                var header1:SOAPHeader=new SOAPHeader(q1,
{string:"bologna",int:"123"});
                var header2:SOAPHeader=new SOAPHeader(q1,
{string:"salami",int:"321"});
                  // Add the header1 SOAP Header to all web service request.
                ws.addHeader(header1);
               // Add the header2 SOAP Header to the getSomething operation.
                ws.getSomething.addHeader(header2);

                // Within the addSimpleHeader method, which adds a SOAP
header to all
              // web service requests, create SOAPHeader and QName objects.
                ws.addSimpleHeader("header3","http://soapinterop.org/xsd",
"foo", "bar");
            }

            // Clear SOAP headers added at the WebService and Operation
levels.
            private function clear():void {
                ws.clearHeaders();
                ws.getSomething.clearHeaders();
            }
        ]]>
    </mx:Script>
```

```
    <mx:HBox>
            <mx:Button label="Clear headers and run again" click="clear();"/
>
    </mx:HBox>

</mx:Application>
```

Redirecting a web service to a different URL

Some web services require that you change to a different endpoint URL after you process the WSDL and make an initial call to the web service. For example, suppose you want to use a web service that requires you to pass security credentials. After you call the web service to send login credentials, it accepts the credentials and returns the actual endpoint URL that is required to use the service's business operations. Before calling the business operations, you must change the `endpointURI` property of your WebService component.

The following example shows a result event listener that stores the endpoint URL that a web service returns in a variable, and then passes that variable into a function to change the endpoint URL for subsequent requests:

```
...
public function onLoginResult(event:ResultEvent):void {

//  Extract the new service endpoint from the login result.
    var newServiceURL = event.result.serverUrl;

// Redirect all service operations to the URL received in the login result.
  serviceName.endpointURI=newServiceURL;

}
...
```

A web service that requires you to pass security credentials, might also return an identifier that you must attach in a SOAP header for subsequent requests; for more information, see "Working with SOAP headers" on page 1447.

CHAPTER 46

Configuring RPC Services

46

You connect to a remote procedure call (RPC) service from an Adobe Flex client application by declaring an RPC component in MXML or ActionScript. The RPC component can specify the actual URL or WSDL URL of an HTTP service or web service, or the name of a Flex Data Services destination definition. A *destination* is the actual server-side service or object that you want to call. You configure destinations in the Flex services configuration file or a file that it includes by reference. This topic describes how to configure and secure destinations.

For information about specifying a destination in an RPC component, see Chapter 45, "Using RPC Components," on page 1407.

Contents

Understanding destination configuration

An RPC service *destination* is the object or service that you connect to using an `<mx:RemoteObject>`, `<mx:WebService>`, or `<mx:HTTPService>` tag or the corresponding ActionScript API. Configuring destinations is the most common task that you perform in the services-config.xml file or a files that it includes by reference. You define remote object destinations in *Remoting Service* definitions; by convention, these are in the remoting-config.xml file. You define web service and HTTP service destinations in *Proxy Service* definitions; by convention, these are in the proxy-config.xml file.

When you configure a destination, you reference one or more messaging channels that you can use to contact the corresponding server-side object or service. You also reference one or more adapters. An *adapter* is server-side code that interacts directly with the object or service. You can configure default adapters to avoid explicitly referencing them in each destination.

The following example shows a basic Remoting Service definition. The service contains a destination that references a security constraint, which is also shown. The destination uses a default adapter, `java-object`, defined at the service level.

```
<service id="remoting-service"
  class="flex.messaging.services.RemotingService"
  messageTypes="flex.messaging.messages.RemotingMessage">

  <adapters>
    <adapter-definition id="java-object"
      class="flex.messaging.services.remoting.adapters.JavaAdapter"
      default="true"/>
  </adapters>

  <default-channels>
    <channel ref="samples-amf"/>
  </default-channels>

  <destination id="SampleEmployeeRO">
    <properties>
      <source>samples.explorer.EmployeeManager</source>
      <scope>application</scope>
    </properties>
    <security>
      <security-constraint ref="privileged-users"/>
    </security>
  </destination>
</service>

...
<security>
  <security-constraint id="privileged-users">
    <auth-method>Custom</auth-method>
    <roles>
      <role>privilegedusers</role>
      <role>admins</role>
    </roles>
  </security-constraint>
...
</security>
```

Message channels

A destination references one or more message channels, which are defined elsewhere in the same configuration file. Optionally, you can reference default channels at the service level in the configuration file instead of referencing them implicitly in specific destinations.

For more information about messaging channels, see "Configuring message channels" on page 1360.

Destination adapters

A service definition includes definitions for one or more adapters that act on a service request. For example, an HTTP service sends HTTP request messages to the Proxy Service, which uses its HTTP proxy adapter to make an HTTP request for a given URL. An adapter definition must reference an adapter class that is an implementation of flex.messaging.services.ServiceAdapter.

You can configure a default adapter to avoid explicitly referencing an adapter in each destination definition. To make an adapter into a default adapter, you set the value of its default property to `true`.

The following example shows a Java object adapter that you use with Remoting Service destinations. This adapter is configured as a default adapter.

```
...
<adapters>
  <adapter-definition id="java-object"
    class="flex.messaging.services.remoting.adapters.JavaAdapter"
    default="true"/>
</adapters>
...
```

Security

You use a *security constraint* to authenticate and authorize users before allowing them to access a destination. You can specify whether to use basic or custom authentication, and indicate the roles that are required for authorization.

You can declare a security constraint inline in a destination definition, or you can declare it globally and reference it by its `id` in a destination definition.

For more information about security, see "Securing destinations" on page 1379.

Default HTTP service destination

You can configure a default destination for HTTP services in the Flex service configuration file or a file that it includes by reference. This lets you use more than one HTTP service `url` property value in client-side service tags or ActionScript code, and still go through the Proxy Service for cross-domain support and security.

The default destination always has an `id` value of `defaultHTTP`. Typically, you use `dynamic-url` parameters to specify one or more URL wildcard patterns for the HTTP proxy adapter.

The following example shows a default destination definition that specifies a `dynamic-url` value:

```
...
<destination id="defaultHTTP">
  <channels>
    <channel ref="my-amf"/>
    </channels>
  <properties>
      <dynamic-url>http://mysite.com/myservices/*</dynamic-url>
  </properties>
  <security>
...
  </security>
</destination>
...
```

For more information about HTTP service adapters, see "HTTP service properties" on page 1456.

Configuring destination properties

To communicate with Remoting Service and Proxy Service destinations, you configutre specific types of destination properties in the `properties` section of a destination definition.

Remote object properties

You use the `source` and `scope` elements of a Remoting Service destination definition to specify the Java object that the destination uses and whether it should be available in the request scope (stateless), the application scope, or the session scope. The following table describes these properties:

Element	Description
`source`	Fully qualified class name of the Java object (remote object).
`scope`	Indicates whether the object is available in the request scope, the application scope, or the session scope. Objects in the request scope are stateless. Objects in the application scope are available to the web application that contains the object. Objects in the session scope are available to the entire client session. The valid values are `request`, `application`, and `session`. The default value is `request`.

The following example shows a Remoting Service destination definition that contains `source` and `scope` properties:

```
...
  <destination id="SampleEmployeeRO">
    <properties>
      <source>samples.explorer.EmployeeManager</source>
      <scope>application</scope>
    </properties>
    <adapter ref="java-object" />
  </destination>
...
```

Web service properties

You use the `wsdl` and `soap` elements to configure web service URLs. These elements define which URLs are permitted for a destination. The following table describes those elements:

Element	Description
`wsdl`	(Optional) Default WSDL URL.
`soap`	SOAP endpoint URL patterns that would typically be defined for each operation in the WSDL document. You can use more than one soap entry to specify multiple SOAP endpoint patterns. Flex matches these values against `endpointURI` property values that you specify in client-side service tags or ActionScript code.

The following example shows a destination definition for a web service:

```
...
  <destination id="ContactManagerWS">
    <adapter ref="soap-proxy" />
    <properties>
      <wsdl>{context.root}/services/ContactManagerWS?wsdl</wsdl>
      <soap>{context.root}/services/ContactManagerWS</soap>
    </properties>
  </destination>
...
```

HTTP service properties

You use the `url` and `dynamic-url` elements to configure HTTP service URLs. These elements define which URLs are permitted for a destination. The following table describes those elements:

Element	Description
`url`	(Optional) Default URL.
`dynamic-url`	(Optional) HTTP service URL patterns. You can use more than one dynamic-url entry to specify multiple URL patterns. Flex matches these values against `url` property values that you specify in client-side service tags or ActionScript code.

The following example shows a destination definition for an HTTP service:

```
...
  <destination id="samplesProxy">
    <channels>
      <channel ref="samples-amf"/>
    </channels>

    <properties>
      <dynamic-url>{context.root}/photoviewer/*</dynamic-url>
      <dynamic-url>{context.root}/blogreader/*</dynamic-url>
      <dynamic-url>{context.root}/services/*</dynamic-url>
    </properties>
  </destination>
...
```

Configuring the Proxy Service

The Proxy Service, in which you define both web service and HTTP service destinations, contains a `properties` element with child elements for configuring the Apache connection manager, self-signed certificates for SSL, and external proxies. The following table describes these XML elements:

Element	Description
`connection-manager`	Contains the `max-connections` and `default-max-connections-per-host` elements. The `max-connections` element controls the maximum total number of concurrent connections that the proxy supports. If the value is greater than `0`, Flex Data Services uses a multithreaded connection manager for the underlying Apache HttpClient proxy. The `default-max-connections-per-host` element sets the default number of connections allowed for each host in an environment that uses hardware clustering.
`content-chunked`	Indicates whether to use chunked content. The default values is `false`. Flash Player does not support chunked content.
`allow-lax-ssl`	Allows self-signed certificates when using SSL, when set to `true`. Do not set the value to `true` in a production environment.
`external-proxy`	Specifies the location of an external proxy as well as a user name and a password for situations where the Proxy Service must contact an external proxy before getting access to the Internet.

The following example shows `max-connections` and `external-proxy` configuration:

```
...
<service id="proxy-service"
  class="flex.messaging.services.HTTPProxyService"
  messageTypes="flex.messaging.messages.HTTPMessage,flex.messaging.
  messages.SOAPMessage">
...

  <properties>
    <connection-manager>
      <max-total-connections>100</max-total-connections>
      <default-max-connections-per-host>2
      </default-max-connections-per-host>
       </connection-manager>
```

```
<!-- Allow self-signed certificates; should not be used in production -->
     <allow-lax-ssl>true</allow-lax-ssl>

     <external-proxy>
       <server>10.10.10.10</server>
       <port>3128</port>
       <nt-domain>mycompany</nt-domain>
       <username>flex</username>
       <password>flex</password>
       </external-proxy>
     </properties>
...
</service>
```

CHAPTER 47

Understanding Flex Messaging

47

Adobe Flex Data Services include a client-side messaging API that you use in conjunction with the server-side Message Service. This topic provides a general introduction to messaging and the Adobe Flex messaging feature.

Contents

About messaging

The Flex messaging feature is based on established messaging standards and terminology. Flex messaging provides a client-side API and a corresponding server-side Message Service (Flex Message Service) for creating Flex messaging applications.Flex messaging also enables participation in Java Message Service (JMS) messaging. For more information, see Chapter 49, “Configuring the Message Service,” on page 1483.

You can also send messages to a ColdFusion Component (CFC) by using the ColdFusion Event Gateway Adapter. For more information, see the ColdFusion Event Gateway Adapter documentation.

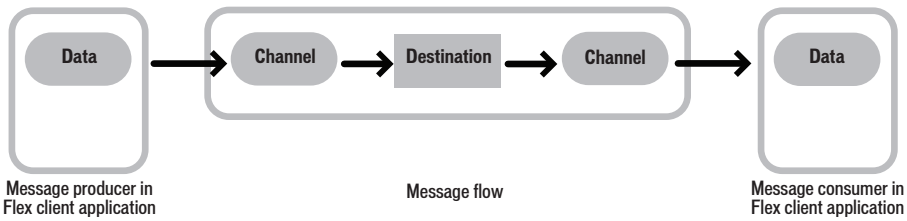
Messaging systems let separate applications communicate asynchronously as peers by passing packets of data called *messages* back and forth through a message service. A message usually consists of a header and a body. The header contains an identifier and routing information. The body contains application data.

Applications that send messages are called *producers*. Applications that receive messages are called *consumers*. In most messaging systems, producers and consumers do not need to know anything about each other. Producers send messages to specific message destinations, and the message service routes the messages to the appropriate consumers.

A *message channel* connects producers and consumers to message destinations. To send messages over a particular channel, an application connects to the message endpoint associated with the message channel. A *message endpoint* is the code responsible for encoding data into messages, and decoding messages into a format that consumers can use. In some messaging systems, an endpoint can pass decoded messages to a message broker, which routes them to appropriate destinations.

A *message adapter* is code that acts as a conduit between the Flex Message Service and other messaging systems. For example, the Java Message Service (JMS) adapter is a message adapter that lets Flex applications subscribe to JMS topics and queues. This adapter lets a pure Java JMS message application share the same destination as a Flex application: Java applications can publish messages to Flex, and Java code can respond to messages that Flex applications send.

The following image shows the flow of data from a message producer to a message consumer. Data is encoded as a message and sent over a channel to a destination. The message is then sent over a channel to the message consumer and decoded into data that the consumer can use.



Flex supports two types of messaging: publish-subscribe messaging, also known as topic-based messaging, and point-to-point messaging, also known as queue-based messaging. These are the two most common types of messaging that enterprise messaging systems use. You use publish-subscribe messaging for applications that require a one-to-many relationship between producers and consumers. You use point-to-point messaging for applications that require a one-to-one relationship between producers and consumers.

In Flex messaging, a topic or queue is represented by a message service destination. In publish-subscribe messaging, each message can have multiple consumers. You use this type of messaging when you want more than one consumer to receive the same message. Examples of applications that might use publish-subscribe messaging are auction sites, stock quote services, and other applications that require one message to be sent to many subscribers. In queue-based messaging, each message is delivered to a single consumer.

Producers publish messages to specific topics on a message server, and consumers subscribe to those topics to receive messages. Consumers can consume messages that were published to a topic only after they subscribed to the topic.

The following image shows a simple publish-subscribe message flow:

Using messaging in an application

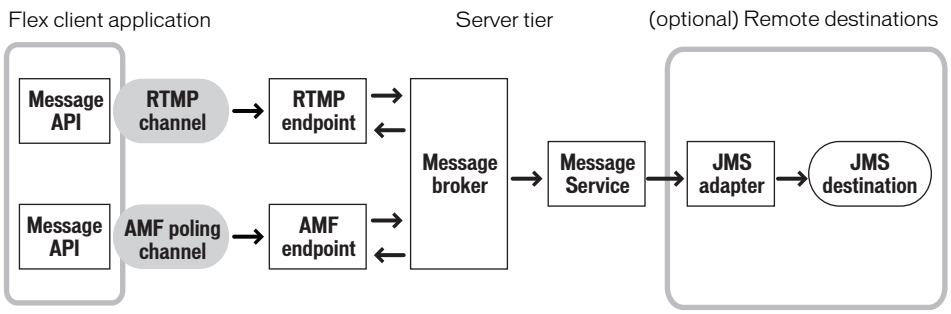
Flex provides MXML and ActionScript APIs that let you add messaging to Flex applications. You can create applications that act as producers, consumers, or both. Flex applications send messages over channels declared on the Flex server to destinations also declared on the Flex server. For more information, see "Configuring message channels" on page 1360.

You configure channels and destinations for the message service in the Flex services configuration file. A message destination is a remote resource; you can configure its security policy and the messaging adapters that it requires. For more information about messaging configuration, see Chapter 49, "Configuring the Message Service," on page 1483.

Understanding the Flex messaging architecture

Flex messaging lets a Flex application connect to a message destination, send messages to it, and receive messages from other messaging clients. Those messaging clients can be Flex applications or other types of clients, such as Java Message Service (JMS) clients. JMS is a Java API that lets applications create, send, receive, and read messages. JMS clients can publish and subscribe to the same message destinations as Flex applications. This means that Flex applications can exchange messages with Java client applications. However, you can create a wide variety of messaging applications using just Flex messaging.

The following image shows a simplified view of the messaging architecture:



Flex Message Service

The four components of the Flex Message Service are channels, destinations, producers, and consumers. When a Flex client application uses the ActionScript or MXML messaging API to publish a message to a destination, the client application sends the message to the server-side Message Service.

The Flex Message Service provides an abstraction on top of the transport protocols that Adobe Flash Player supports, such as Action Message Format (AMF), Realtime Message Protocol (RTMP), and XMLSocket. You configure the Message Service to transport messages over one or more channels. Each channel corresponds to a specific transport protocol. Within the Flex services configuration file, you specify which channels to use for a specific destination.

Message channels

Flex applications can access the Flex Message Service over several different message channels. Each channel corresponds to a specific network protocol and has a corresponding server-side endpoint. There is client-side code that corresponds to each channel. When a message arrives at an endpoint, the endpoint decodes the message and passes it to a message broker, which determines where the message should be sent. For publish-subscribe messaging scenarios, the message broker sends messages to the Message Service. The message broker also routes messages for the Flex remote procedure call (RPC) service and Data Management Service features. The Flex client tries the channels in the order specified until an available channel is found or there are no more channels in the list.

For more information about message channels, see “Message channels” on page 1485.

JMS message adapter

JMS is a Java API that lets applications create, send, receive, and read messages. Flex messaging uses the JMS message adapter to interact with external messaging systems.

This adapter lets Flex applications subscribe to JMS topics and queues. It allows Flex applications to participate in existing messaging oriented middleware (MOM) systems.

CHAPTER 48

Using Flex Messaging

48

Adobe Flex Data Services includes a client-side messaging API that you use in conjunction with the server-side Flex Message Service. This topic describes how to use the client-side messaging API to create Flex applications that send and receive messages. For information about configuring the server-side Message Service, see Chapter 49, "Configuring the Message Service," on page 1483.

Contents

Using messaging in a Flex application

A Flex client application uses the client-side messaging API to send messages to, and receive messages from, a server-side destination. Messages are sent to and from destinations over a protocol-specific message channel. er

The two primary client-side messaging components are Producer and Consumer components. A Producer component sends messages to a server-side destination. A Consumer component subscribes to a server-side destination and receives messages that a Producer component sends to that destination. You can create Producer and Consumer components in MXML or ActionScript.

Producer and Consumer components both require a valid message destination that you configure in the Flex services configuration file. For information about message destinations, see Chapter 49, "Configuring the Message Service," on page 1483.

A Flex application often contains at least one pair of Producer and Consumer components. This enables each application to send messages to a destination and receive messages that other applications send to that destination. It is very easy to test this type of application by running it in different browser windows and sending messages from each of them.

Working with Producer components

You can create Producer components in MXML or ActionScript. To send a message, you create an AsyncMessage object and then call a Producer component's `send()` method to send the message to a destination.

You can also specify acknowledge and fault event handlers for a Producer component. An acknowledge event is broadcast when a destination successfully receives a message that a Producer component sends. A fault event is dispatched when a destination cannot successfully process a message due to a connection-, server-, or application-level failure.

A Producer component can send messages to destinations that use the Flex Message Service with no message adapters or to destinations that use message adapters. A *message adapter* is server-side code that acts as a conduit between the Flex Message Service and other messaging systems. For example, the Java Message Service (JMS) adapter enables Producer components to send messages to JMS topics. On the client side, the APIs for non-adapter and adapter-enabled destinations are identical. The definition for a particular destination in the Flex services configuration file determines what happens when a Producer component sends messages to the destination or a Consumer component subscribes to the destination. For information about configuring destinations, see Chapter 49, "Configuring the Message Service," on page 1483.

For reference information about the Producer class, see *Adobe Flex 2 Language Reference*.

Creating a Producer component in MXML

You use the `<mx:Producer>` tag to create a Producer component in MXML. The tag must contain an `id` value. It should also specify a `destination` that is defined in the server-side services-config.xml file.

The following code shows an <mx:Producer> tag that specifies a destination and acknowledge and fault event handlers:

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateProducerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.rpc.events.FaultEvent;
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
           private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
        ]]>
    </mx:Script>
    <mx:Producer id="producer"
        destination="ChatTopicJMS"
        acknowledge="acknowledgeHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

Creating a Producer component in ActionScript

You can create a Producer component in an ActionScript method. The following code shows a Producer component that is created in a method in an `<mx:Script>` tag. The import statements import the classes required to create a Producer component, create Message objects, add event listeners, and create message handlers.

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateProducerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private var producer:Producer;
           private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void{
            // Handle fault event.
            }
            private function logon():void {
                producer = new Producer();
                producer.destination = "ChatTopicJMS";
                producer.addEventListener(MessageAckEvent.ACKNOWLEDGE,
acknowledgeHandler);
                producer.addEventListener(MessageFaultEvent.FAULT,
faultHandler);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Sending a message to a destination

To send a message from a Producer component to a destination, you create an mx.messaging.messages.AsyncMessage object, populate the body of the AsyncMessage object, and then call the component's `send()` method. You can create text messages and messages that contain objects.

The following code creates a message, populates the body of the message with text, and sends the message by calling a Producer component's `send()` method:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private var producer:Producer;

            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body = userName.text + ": " + input.text;
                producer.send(message);
            }
        ]]>
    </mx:Script>
    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send" click="sendMessage();"/>
</mx:Application>
```

The following code creates a message, populates the body of the message with an object, and sends the message by calling a Producer component's `send()` method. The `TypedObject()` method refers to the constructor of an ActionScript object named TypedObject that is available in the source path.

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateProducerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.rpc.events.FaultEvent;
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
           private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
        ]]>
    </mx:Script>
    <mx:Producer id="producer"
        destination="ChatTopicJMS"
        acknowledge="acknowledgeHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

Adding extra information to a message

You can include extra information for a message in the form of message headers. You can send strings and numbers in message headers. The headers of a message are contained in an associative array where the key is the header name. The `headers` property of a message class lets you add headers for a specific message instance.

The following code adds a message header called prop1 and sets its value:

```
<?xml version="1.0"?>
<!-- fds\messaging\SendMessageHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private var producer:Producer;

            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.headers = new Array();
                message.headers["prop1"] = 5;
                message.body =input.text;
                producer.send(message);
            }
        ]]>
    </mx:Script>
    <mx:TextInput id="input"/>
</mx:Application>
```

You can use a Consumer component's `selector` property to filter the messages that the component receives, based on message header values. For more information, see "Filtering messages with a message selector" on page 1474.

NOTE	Do not start message header names with the text JMS or DS. These prefixes are reserved.

Resending messages and timing out requests

A Producer component sends a message once. If the delivery of a message that a Producer component sends is in doubt, the Producer component dispatches a fault event, which indicates that it never received an acknowledgment for the specified message. When the event is dispatched, the handler code can then make a decision to abort or attempt to resend the faulted message. Two events can trigger a fault that indicated delivery is in doubt. It can be triggered when the value of the `requestTimeout` property is exceeded or the underlying message channel becomes disconnected before the acknowledgment message is received. The fault handler code can detect this scenario by inspecting the `faultCode` property of the associated ErrorMessage for the ErrorMessage.MESSAGE_DELIVERY_IN_DOUBT code.

Working with Consumer components

You can create Consumer components in MXML or ActionScript. To subscribe to a destination, you call a Consumer component's `subscribe()` method.

You can also specify message and fault event handlers for a Consumer component. A message event is broadcast when a message is received by the destination to which a Consumer component is subscribed. A fault event is broadcast when the channel to which the Consumer component is subscribed can't establish a connection to the destination, the subscription request is denied, or if a failure occurs when the Consumer component's `receive()` method is called. The `receive()` method retrieves any pending messages from the destination.

For reference information about the Consumer class, see *Adobe Flex 2 Language Reference*.

Creating a Consumer component in MXML

You use the `<mx:Consumer>` tag to create a Consumer component in MXML. The tag must contain an `id` value. It should also specify a `destination` that is defined in the server-side services-config.xml file.

The following code shows an `<mx:Consumer>` tag that specifies a destination and acknowledge and fault event handlers:

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateConsumerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
        ]]>
    </mx:Script>
    <mx:Consumer id="consumer"
        destination="ChatTopicJMS"
        message="messageHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

Creating a Consumer component in ActionScript

You can create a Consumer component in an ActionScript method. The following code shows a Consumer component created in a method in an `<mx:Script>` tag. The import statements import the classes required to create a Consumer component, create AsyncMessage objects, add event listeners, and create message handlers.

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateConsumerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private var consumer:Consumer;
           private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void{
            // Handle fault event.
            }
            private function logon():void {
                consumer = new Consumer();
                consumer.destination = "ChatTopicJMS";
                consumer.addEventListener
                    (MessageAckEvent.ACKNOWLEDGE,  acknowledgeHandler);
                consumer.addEventListener
                    (MessageFaultEvent.FAULT, faultHandler);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Subscribing to a destination

Whether you create a Consumer component in MXML or ActionScript, you still must call the component's `subscribe()` method to subscribe to a destination and receive messages from that destination.

A Consumer component can subscribe to destinations that use the Flex Message Service with no message adapters or to destinations that use message adapters. For example, the Java Message Service (JMS) adapter enables Consumer components to subscribe to JMS topics. On the client side, the API for non-adapter and adapter-enabled destinations is identical. The definition for a particular destination in the Flex services configuration file determines what happens when a Consumer component subscribes to the destination or a Producer component sends messages to the destination. For information about configuring destinations, see Chapter 49, "Configuring the Message Service," on page 1483.

The following code shows a call to a Consumer component's `subscribe()` method:

```
<?xml version="1.0"?>
<!-- fds\messaging\Subscribe.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private var consumer:Consumer;

            private function logon():void {
                consumer = new Consumer();
                consumer.subscribe();
            }
        ]]>
    </mx:Script>
</mx:Application>
```

You can unsubscribe a Consumer component from a destination by calling the component's `unsubscribe()` method.

Filtering messages with a message selector

You can use a Consumer component's `selector` property to filter the messages that the component should receive. A message selector is a String that contains a SQL conditional expression based on the SQL92 conditional expression syntax. The Consumer component receives only messages with headers that match the selector criteria. For information about creating message headers, see "Adding extra information to a message" on page 1470.

The following code sets a Consumer component's `selector` property in an `<mx:Consumer>` tag:

```
<?xml version="1.0"?>
<!-- fds\messaging\CreateConsumerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
        ]]>
    </mx:Script>
    <mx:Consumer id="consumer"
        destination="ChatTopicJMS"
        selector="prop1 > 5"
        message="messageHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

The following code sets a Consumer component's `selector` property in ActionScript:

```
<?xml version="1.0"?>
<!-- fds\messaging\SelectorAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.events.*;
            private var consumer:Consumer;

            private function logon():void {
                consumer = new Consumer();
                consumer.destination = "ChatTopic";
                consumer.selector = "prop1 > 5";
                consumer.subscribe();
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Using subtopics

The subtopic feature lets you divide the messages that a Producer component sends to a destination into specific categories at the destination. You can configure a Consumer component that subscribes to the destination to receive only messages sent to a specific subtopic or set of subtopics. You use wildcard characters (*) to send or receive messages from more than one subtopic.

You cannot use subtopics with a JMS destination. However, you can use message headers and Consumer selector expressions to achieve similar functionality when using JMS. For more information, see "Filtering messages with a message selector" on page 1474.

In the `subtopic` property of a Producer component, you specify the subtopic(s) that the component sends messages to. In the `subtopic` property of a Consumer component, you specify the subtopic(s) to which the Consumer is subscribed.

To send a message from a Producer component to a destination and a subtopic, you set the `destination` and `subtopic` properties, and then call the `send()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- fds\messaging\Subtopic1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

           private function acknowledgeHandler(event:MessageAckEvent):void
{
                // Handle message acknowledgement event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
            private function useSubtopic():void {
                var message:AsyncMessage = new AsyncMessage();
                producer.subtopic = "chat.fds.newton";
                // Generate message.
                producer.send(message);
            }
        ]]>
    </mx:Script>
    <mx:Producer id="producer"
        destination="ChatTopicJMS"
        acknowledge="acknowledgeHandler(event)"
        fault="faultHandler(event)"/>
</mx:Application>
```

To subscribe to a destination and a subtopic with a Consumer component, you set the `destination` and `subtopic` properties and then call the `subscribe()` method, as the following example shows. This example uses a wildcard character (*) to receive all messages sent to all subtopics under the chat.fds subtopic.

```
<?xml version="1.0"?>
<!-- fds\messaging\Subtopic2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
            }
            private function faultHandler(event:MessageFaultEvent):void {
            // Handle fault event.
            }
            private function useSubtopic():void {
                consumer.destination = "ChatTopic";
                consumer.subtopic = "chat.fds.*";
                consumer.subscribe();
            }
        ]]>
    </mx:Script>
    <mx:Consumer id="consumer" destination="ChatTopicJMS"
         message="messageHandler(event)"
         fault="faultHandler(event)"/>
</mx:Application>
```

To allow subtopics for a destination, you must set the `allow-subtopics` element to `true` in the destination definition in the services-config.xml file or a file that it includes by reference, such as the messaging-config.xml file. The `subtopic-separator` element is optional; the default value is `"."` (period).

```
<destination id="ChatTopic">
  <properties>
    <network>
      <session-timeout>0</session-timeout>
    </network>
    <server>
      <max-cache-size>1000</max-cache-size>
      <message-time-to-live>0</message-time-to-live>
      <durable>false</durable>
      <allow-subtopics>true</allow-subtopics>
      <subtopic-separator>.</subtopic-separator>
    </server>
  </properties>
```

```
  <channels>
    <channel ref="my-rtmp"/>
  </channels>
</destination>
```

The following example is a runnable application that uses subtopics. In this case, the Producer and Consumer components are declared in MXML tags. The Consumer component uses a wildcard character (*) to receive all messages under the chat.fds subtopic.

```
<?xml version="1.0"?>
<!-- fds\messaging\Subtopic3.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function sendMessage():void {
                myCon.subscribe();
                var message:AsyncMessage = new AsyncMessage();
                message.body = userName.text;
                myPub.send(message);
            }
            private function messageHandler(event: MessageEvent):void {
                ta.text += event.message.body + "\n";
            }
        ]]>
    </mx:Script>

    <mx:Producer id="myPub" destination="mike" subtopic = "chat.fds.blue"/>
    <mx:Consumer id="myCon" destination="mike" subtopic="chat.fds.*"
        message="messageHandler(event);"/>


    <mx:TextInput id="userName" width="20%"/>
    <mx:TextInput id="msg" width="20%"/>
    <mx:Button label="Send" click="sendMessage();"/>
    <mx:TextArea id="ta"/>
</mx:Application>
```

Using a pair of Producer and Consumer components in an application

A Flex application often contains at least one pair of Producer and Consumer components. This enables each application to send messages to a destination and receive messages that other applications send to that destination.

To act as a pair, Producer and Consumer components in an application must use the same message destination. Producer component instances send messages to a destination, and Consumer component instances receive messages from that destination.

The following code shows excerpts from a simple chat application that contains a pair of Producer and Consumer components. The user types messages in a `msg` TextInput control; the Producer component sends the message when the user presses the keyboard Enter key or clicks the Button control labeled Send. The user views messages from other users in the `ta` TextArea control. The user can switch between non-JMS and JMS-enabled destinations by selecting items in the topics List control.

```
<?xml version="1.0"?>
<!-- fds\messaging\ProducerConsumer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function logon():void {
                producer.destination = topics.selectedItem.toString();
                consumer.destination = topics.selectedItem.toString();
                consumer.subscribe();
                topics.selectedIndex = 1;
            }

            private function messageHandler(event: MessageEvent):void {
                ta.text += event.message.body + "\n";
            }

            private function sendMessage():void {
                var message: AsyncMessage = new AsyncMessage();
                message.body = userName.text + ": " + msg.text;
                producer.send(message);
                msg.text = "";
            }

        ]]>
    </mx:Script>
```

```
    <mx:Producer id="producer" destination="ChatTopicJMS"/>
    <mx:Consumer id="consumer" destination="ChatTopicJMS"
        message="messageHandler(event)"/>

    <mx:List id="topics">
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:String>chat-topic</mx:String>
                <mx:String>chat-topic-jms</mx:String>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:List>

    <mx:TextArea id="ta" width="100%" height="100%"/>
    <mx:TextInput id="userName" width="100%"/>
    <mx:TextInput id="msg" width="100%"/>
    <mx:Button label="Send" click="sendMessage()"/>
</mx:Application>
```

NOTE

The Flex sample applications include the code for the complete chat application.

CHAPTER 49

Configuring the Message Service

49

To enable messaging in an Adobe Flex client application, you connect to the Flex Message Service by declaring a connection to a server-side destination in `<mx:Producer>` and `<mx:Consumer>` tags or the corresponding ActionScript API. A Message Service *destination* is the endpoint that you send messages to and receive messages from when performing publish-subscribe or point-to-point messaging. You configure destinations as part of the Message Service definition in the Flex services configuration file. This topic describes how to configure destinations.

For information about using the client-side messaging API and connecting to a destination in MXML or ActionScript, see Chapter 48, "Using Flex Messaging," on page 1465.

Contents

Understanding Message Service configuration

The most common tasks that you perform when configuring the Message Service are defining message destinations, applying security to message destinations, and modifying logging settings. A *Message Service destination* is the server-side code that you connect to using Producer and Consumer components. You configure Message Service destinations in the Message Service section of the Flex services configuration file or a file that it includes by reference. By default, the Flex services configuration file is named services-config.xml and is located in the WEB_INF/flex directory of the web application that contains Adobe Flex Data Services. In the Flex service configuration files that Adobe ships, individual configuration files for Flex Data Services, the Flex Message Service, and remote procedure call (RPC) services are usually referenced.

The following example shows a basic Message Service configuration in the Flex services configuration file. It contains one destination that is configured for a chat application that uses the default message adapter, which in this example is the ActionScript adapter.

```
...
<service id="message-service"
  class="flex.messaging.services.MessageService"
  messageTypes="flex.messaging.messages.AsyncMessage">
  <adapters>
    <adapter-definition id="actionscript"
    class="flex.messaging.services.messaging.
      adapters.ActionScriptAdapter" default="true"/>
    <adapter-definition id="jms"
      class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
   </adapters>
...
  <destination id="chat-topic">
    <properties>
      <network>
        <session-timeout>0</session-timeout>
        <throttle-inbound policy="ERROR" max-frequency="50"/>
        <throttle-outbound policy="REPLACE" max-frequency="500"/>
      </network>
      <server>
        <max-cache-size>1000</max-cache-size>
        <message-time-to-live>0</message-time-to-live>
        <durable>true</durable>
        <durable-store-manager>
          flex.messaging.durability.FileStoreManager
        </durable-store-manager>
      </server>
    </properties>
    <channels>
      <channel ref="samples-rtmp"/>
      <channel ref="samples-amf-polling"/>
    </channels>
  </destination>
...
</service>
...
```

Message Service destinations

When you define a destination, you reference one or more message channels that transport messages. You also reference a message adapter or use an adapter that is configured as the default adapter. The ActionScript adapter lets you use messaging with Flex clients as the sole producers and consumers of the messages. The Java Message Service (JMS) message adapter lets you interact with a Java Message Service (JMS) implementation. For more information, see "Message Service adapters" on page 1485.

You also set network- and server-related properties. You can also reference or define security constraints for a destination. For more information, see "Configuring Message Service destinations" on page 1486.

Message channels

A destination references one or more message channels, which are defined in the services-config.xml configuration file. The two most common channels that you use for messaging are the Realtime Message Protocol (RTMP) channel and the Action Message Format (AMF) channel with message polling enabled.

The RTMP channel maintains a connection between the client and the server, so the client does not need to poll the server. The AMF channel with polling enabled polls the server for new messages.

For more information about securing messaging channels, see "Securing destinations" on page 1379.

Message Service adapters

A *Message Service adapter* is the server-side code that facilitates messaging when your application uses ActionScript objects only or interacts with another system, such as an implementation of the Java Message Service (JMS). You reference adapters in a destination definition. You can also specify adapter-specific settings. For more information about message adapters, see "Configuring Message Service destinations" on page 1486.

Security

One way to secure a destination is by using a *security constraint*, which defines the access privileges for the destination.

You use a security constraint to authenticate and authorize users before allowing them to access a destination. You can specify whether to use basic or custom authentication, and indicate the roles required for authorization.

You can declare a security constraint inline in a destination definition, or you can declare it globally and reference it by its `id` in a destination definitions.

For more information about security, see "Securing destinations" on page 1379.

Configuring Message Service destinations

This section provides information about creating Message Service destinations.

Referencing message channels

Messages are transported to and from a destination over a message channel. A destination references one or more message channels that are also defined in the fds.config.xml configuration file. The destination can be contacted using only one of the listed channels.

Any attempt to contact the destination on an unlisted channel results in an error. Flex applications use this configuration information to determine which channels to use to contact the specified destination. The Flex application attempts to use the referenced channels in the order specified. You can also configure default message channels, in which case you do not have to explicitly reference any channels in the destination.

The following example shows a destination's channel references. Because the `samples-rtmp` channel is listed first, the destination attempts to use it first.

```
...
  <destination id="chat-topic">
...
    <channels>
      <channel ref="samples-rtmp"/>
      <channel ref="samples-amf-polling"/>
    </channels>
...
  </destination>
...
```

For more information about message channels, see "Securing destinations" on page 1379.

Setting network properties

A destination contains a set of properties for defining client-server messaging behavior. The following example shows the network-related properties of a destination:

```
...
  <destination id="chat-topic">
    <properties>
      <network>
        <session-timeout>0</session-timeout>
        <throttle-inbound policy="ERROR" max-frequency="50"/>
        <throttle-outbound policy="REPLACE" max-frequency="500"/>
      </network>
  ...
    </properties>
  </destination>
...
```

Message Service destinations use the following network-related properties:

Property	Description
`session-timeout`	Idle time in minutes before a subscriber is unsubscribed. When the value is set to 0 (zero), subscribers are not forced to unsubscribe automatically.
`throttle-inbound`	The `max-frequency` attribute controls how many messages per second the server can receive. The `policy` attribute indicates what to do when the message limit is reached. A `policy` value of `ERROR` indicates that an error should be returned if the limit is reached. A `policy` value of `IGNORE` indicates that no error should be returned if the limit is reached.
`throttle-outbound`	The `max-frequency` attribute controls how many messages per second the server can send. The `policy` attribute indicates what to do when the message limit is reached. A `policy` value of `ERROR` indicates that an error should be returned if the limit is reached. A `policy` value of `IGNORE` indicates that no error should be returned if the limit is reached. A `policy` value of `REPLACE` indicates that the previous message should be replaced when the limit is reached.

Setting server properties

A destination contains a set of properties for controlling server-related parameters. The following example shows server-related properties of a destination:

```
...
  <destination id="chat-topic">
```

```
    <properties>
...
      <server>
        <max-cache-size>1000</max-cache-size>
        <message-time-to-live>0</message-time-to-live>
        <durable>true</durable>
        <durable-store-manager>
          flex.messaging.durability.FileStoreManager
        </durable-store-manager>
      </server>
    </properties>
  </destination>
...
```

Message Service destinations use the following server-related properties:

Property	Description
`max-cache-size`	Maximum number of messages to maintain in memory cache.
`message-time-to-live`	How many milliseconds that a message is kept on the server. A value of 0 means the message lasts forever.
`durable`	Boolean value that indicates whether messages should be saved in a durable message store to ensure that they survive connection outages and reach destination subscribers. When the JMS adapter is used, it inherits the `durable` value.
`durable-store-manager`	Durable store manager class to use when not using the JMS adapter. By default, the flex.messaging.durability.FileStoreManager, included with Flex Data Services, stores messages in files in the /WEB-INF/flex/message_store/*DESTINATION_NAME* of your Flex web application. You can change this location by setting the `file-store-root` property. **Note:** Using the Flex Data Services FileStoreManager for durable messaging is not cluster-safe.
`batch-write-size`	(Optional) Number of durable message files to write in each batch.
`file-store-root`	(Optional) Location for storing durable message files.
`max-file-size`	(Optional) Maximum file size in kilobytes for durable message files.

Referencing Message Service adapters

To use a Message Service adapter, such as the ActionScript, JMS, or ColdFusion Event Gateway adapter included with Flex Data Services, you reference the adapter in a destination definition or use an adapter configured as the default adapter. In addition to referencing an adapter, you also set its properties in a destination definition; for information about the JMS adapter, see "Configuring the JMS adapter" on page 1489.

The following example shows the definition for the JMS adapter and a reference to it in a destination:

```
...
<service id="message-service"
  class="flex.messaging.services.MessageService"
  messageTypes="flex.messaging.messages.AsyncMessage">
...
  <adapters>
    <adapter-definition id="actionscript"
    class="flex.messaging.services.messaging.
      adapters.ActionScriptAdapter" default="true"/>
    <adapter-definition id="jms"
      class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
   </adapters>

...

  <destination id="chat-topic-jms">
...
    <adapter ref="jms"/>
...
  </destination>
...
</service>
```

Configuring the JMS adapter

You use the JMS adapter to subscribe to JMS topics or queues configured on an implementation of the Java Message Service. This adapter lets Java JMS publishers and subscribers share the same destination as a Flex client application. Java code can publish messages to the Flex application, and Java code can respond to messages that the Flex application publishes.

You configure the JMS adapter individually for the destinations that use it. You must configure the adapter with the proper Java Naming and Directory Interface (JNDI) information and JMS ConnectionFactory information to look up the connection factory in JNDI, create a connection from it, create a session on behalf of the Flex client, and create a publisher and subscribers on behalf of the Flex client.

The following example shows a destination that uses the JMS adapter:

```
...
<destination id="chat-topic-jms">
  <properties>
...
     <jms>
       <destination-type>Topic</destination-type>
       <message-type>javax.jms.TextMessage</message-type>
       <connection-factory>jms/flex/TopicConnectionFactory
       </connection-factory>
       <destination-jndi-name>jms/topic/flex/simpletopic
       </destination-jndi-name>
       <destination-name>FlexTopic
       </destination-name>
       <delivery-mode>NON_PERSISTENT</delivery-mode>
       <message-priority>DEFAULT_PRIORITY</message-priority>
       <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
       <transacted-sessions>false</transacted-sessions>
  <!-- (Optional) JNDI environment. Use when using JMS on a remote JNDI
  server. Used to specify the JNDI environment to access an external JMS
  provider. -->
  <initial-context-environment>
    <property>
      <name>Context.SECURITY_PRINCIPAL</name>
      <value>anonymous</value>
    </property>
    <property>
      <name>Context.SECURITY_CREDENTIALS</name>
      <value>anonymous</value>
    </property>
    <property>
      <name>Context.PROVIDER_URL</name>
      <value>http://{server.name}:1856</value>
    </property>
    <property>
       <name>Context.INITIAL_CONTEXT_FACTORY</name>
       <value>fiorano.jms.runtime.naming.FioranoInitialContextFactory
       </value>
    </property>
  </initial-context-environment>

     </jms>
  </properties>
```

```
...
<adapter ref="jms"/>
</destination>
...
```

The JMS adapter accepts the following configuration properties. For more specific information about JMS, see the Java Message Service specification or your application server documentation.

Property	Description
`destination-type`	(Optional) Type of messaging that the adapter is performing. Valid values are `topic` for publish-subscribe messaging and `queue` for point-to-point messaging. The default value is `topic`.
`message-type`	Type of the message to use when transforming Flex messages into JMS messages. Supported types are javax.jms.TextMessage and javax.jms.ObjectMessage. If the client-side Publisher component sends messages as objects, you must set the message-type to javax.jms.ObjectMessage.
`connection-factory`	Name of the JMS connection factory in JNDI.
`destination-jndi-name`	Name of the destination in the JNDI registry.
`destination-name`	(Optional) Name of the destination in JMS. The default value is the Flex destination `id`.
`delivery-mode`	JMS DeliveryMode for producers. The valid values are `PERSISTENT` and `NON_PERSISTENT`. The `PERSISTENT` mode causes all sent messages to be stored by the JMS server and then forwarded to consumers. This adds processing overhead but is necessary for guaranteed delivery. The `NON_PERSISTENT` mode does not require that messages be stored by the JMS server before forwarding to consumers, so they may be lost if the JMS server fails while processing the message; this setting is suitable for notification messages that do not require guaranteed delivery.
`message-priority`	JMS priority for messages that Flex producers send. The valid values are `DEFAULT_PRIORITY` or an integer value indicating what the priority should be. The JMS API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. Additionally, clients should consider priorities 0-4 as gradations of normal priority, and priorities 5-9 as gradations of expedited priority.

Property	Description
acknowledge-mode	JMS message acknowledgment mode. The valid values are AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE, and CLIENT_ACKNOWLEDGE.
transacted-sessions	JMS session transaction mode.
initial-context-environment	A set of JNDI properties for configuring the InitialContext used for JNDI lookups of your ConnectionFactory and Destination. Lets you use a remote JNDI server for JMS. For more information, see "Using a remote JMS provider" on page 1492.

Using a remote JMS provider

You can use JMS on a remote JNDI server by configuring the optional `initial-context-environment` element in the `jms` section of a message destination that uses the JMS adapter. The `initial-context-environment` element takes `property` subelements, which in turn take `name` and `value` subelements. In the `name` and `value` elements, you can specify javax.naming.Context constant names and corresponding values, or you can specify string literal names and corresponding values to establish the desired JNDI environment.

The boldfaced code in the following example is an `initial-context-environment` configuration:

```
...
<destination id="chat-topic-jms">
  <properties>
...
     <jms>
       <destination-type>Topic</destination-type>
       <message-type>javax.jms.TextMessage</message-type>
       <connection-factory>jms/flex/TopicConnectionFactory
       </connection-factory>
       <destination-jndi-name>jms/topic/flex/simpletopic
       </destination-jndi-name>
       <destination-name>FlexTopic
       </destination-name>
       <delivery-mode>NON_PERSISTENT</delivery-mode>
       <message-priority>DEFAULT_PRIORITY</message-priority>
       <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
       <transacted-sessions>false</transacted-sessions>
  <!-- (Optional) JNDI environment. Use when using JMS on a remote JNDI
  server. -->
  <initial-context-environment>
    <property>
      <name>Context.SECURITY_PRINCIPAL</name>
      <value>anonymous</value>
```

```
    </property>
    <property>
      <name>Context.SECURITY_CREDENTIALS</name>
      <value>anonymous</value>
    </property>
    <property>
      <name>Context.PROVIDER_URL</name>
      <value>http://{server.name}:1856</value>
    </property>
    <property>
      <name>Context.INITIAL_CONTEXT_FACTORY</name>
      <value>fiorano.jms.runtime.naming.FioranoInitialContextFactory</
  value>
    </property>
  </initial-context-environment>

    </jms>
  </properties>
...
<adapter ref="jms"/>
</destination>
...
```

Flex treats `name` element values that begin with the text `Context.` as the constants defined by javax.naming.Context and verifies that the Context class defines the constants that you specify. Some JMS providers also allow custom properties to be set in the initial context, and you can specify these by using the string literal name and corresponding value that the provider requires. For example, the FioranoMQ JMS provider configures failover to backup servers with the following property:

```
<property>
   <name>BackupConnectURLs</name>
   <value>http://backup-server:1856;http://backup-server-2:1856</value>
</property>
```

If you do not specify the `initial-context-environment` properties in the `jms` section of a destination definition, the default JNDI environment is used. The default JNDI environment is configured in a jndiprovider.properties application resource file and or a jndi.properties file.

Depending on the JNDI environment that you are using, the connection-factory and destination-jndi-name configuration elements must correctly reference the target named instances in the directory; naming conventions across JNDI providers for topic connection factories and destinations may vary. You must also include your JMS provider's client library JAR files in the WEB-INF/lib directory of your Flex web application or in another location from which the class loader will load them. Even when using an external JMS provider, Flex Data Services uses the `connection-factory` and `destination-jndi-name` configuration properties to look up the necessary connection factory and destination instances.

Creating a custom Message Service adapter

You can create a custom Message Service adapter for situations where you need functionality that is not provided by one of the standard adapters. A Message Service adapter class must extend the flex.messaging.services.ServiceAdapter class. An adapter calls methods on an instance of a flex.messaging.MessageService object. Both ServiceAdapter and MessageService are included in the public Flex Data Services Javadoc documentation.

The primary method of any Message Service adapter class is the `invoke()` method, which is called when a client sends a message to a destination. Within the `invoke()` method, you can include code to send messages to all subscribing clients or to specific clients by evaluating selector statements included with a message from a client.

To send a message to clients, you call the `MessageService.pushMessageToClients()` method in your adapter's `invoke()` method. This method takes a message object as its first parameter. Its second parameter is a Boolean value that indicates whether to evaluate message selector statements. You can call the `MessageService.sendPushMessageFromPeer()` method in your adapter's `invoke()` method to broadcast messages to peer server nodes in a clustered environment.

```
package customclasspackage;

import flex.messaging.services.ServiceAdapter;
import flex.messaging.services.MessageService;
import flex.messaging.messages.Message;
import flex.messaging.Destination;

public class SimpleCustomAdapter extends ServiceAdapter {

    public Object invoke(Message message) {
        MessageService msgService = (MessageService)service;
        msgService.pushMessageToClients(message, true);
        msgService.sendPushMessageFromPeer(message, true);
        return null;
    }
}
```

Optionally, a Message Service adapter can manage its own subscriptions. To do this, you override the `ServiceAdapter.handlesSubscriptions()` method and return `true`. You also must override the `ServiceAdapter.manage()` method, which is passed CommandMessages for subscribe and unsubscribe operations.

The ServiceAdapter class's `getAdapterState()` and `setAdapterState()` methods are for adapters that maintain an in-memory state that must be replicated across a cluster. When an adapter starts up, it gets a copy of that state from another cluster node when there is another node running.

To use an adapter class, you must specify it in an `adapter-definition` element in the Message Services configuration, as the following example shows:

```
<adapters>
...
  adapter-definition id="cfgateway" class="foo.bar.SampleMessageAdapter"/>
...
</adapters>
```

Another optional feature that you can implement in an adapter is MBean component management. This lets you expose properties for getting and setting in the administration console. For more information, see Chapter 43, "Managing services," on page 1390.

CHAPTER 50

Understanding the Flex Data Management Service

50

The Adobe Flex Data Management Service feature spans the client and server tiers to provide the top-level functionality for distributed data in Flex applications. This feature lets you build applications that provide data synchronization, data replication, and occasionally connected application services. Additionally, you can manage large collections of data and nested data relationships, such as one-to-one and one-to-many relationships, and use Data Management Service adapters to integrate with data resources.

A client-side DataService component, which you can create in MXML or ActionScript, calls methods on a destination configured in the server-side Data Management Service. The client-side component performs activities such as filling client-side objects with data from remote data resources, and synchronizing the versions of data in multiple client instances with the server-side destination. This topic provides a general introduction to the Data Management Service feature.

Contents

About the Data Management Service feature

The Data Management Service feature uses a combination of client-side and server-side functionality to distribute data among multiple clients and the server tier. The client-side DataService component is an ActionScript object that works in conjunction with the server-side Data Management Service to provide distributed data management. The client-side DataService component manages data on the client, and the server-side Data Management Service manages the distribution of data among multiple clients and server-side data resources. The data that a DataService component manages is often replicated through the Data Management Service from a remote data resource or another client application.

Comparing the Data Management Service and RPC features

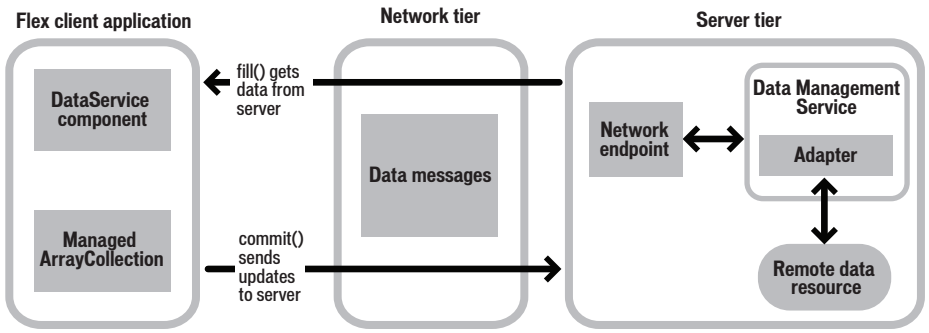
The Data Management Service feature employs an approach to data that is fundamentally different from the remote procedure call (RPC) approach of the Flex RPC feature. Unlike the RPC feature, the Data Management Service feature supports automatic and manual synchronization of a common set of data on multiple clients and server-side data resources. It also supports offline client-side data persistence for occasionally connected clients.

Unlike an RPC component, a DataService component does not invoke operations or methods directly on a remote service that returns static results that you then bind into an object. Instead, a set of data is replicated to multiple clients from the Data Management Service and as data is changed, the changes are automatically set to the Data Management Service, which updates the underlying data resource and client applications.

Understanding the flow of data

At the heart of the Data Management Service feature is the Flex messaging framework, which passes data messages between Flex client applications and the Data Management Service.Data messages are passed across the network tier over a message channel, which connects message producers and consumers to destinations. The Data Management Service supports several different message channels, including channels that support the HTTP, Action Message Format (AMF), and Realtime Message Protocol (RTMP) protocols. Data Management Service client applications act as both message producers and consumers that send data to a Data Management Service destination and subscribe to the destination to receive data updates.

The following image shows the flow of data between a back-end data resource and an ArrayCollection object that a DataService component manages in a Flex application:



The Data Management Service uses a *Data Management Service adapter* to update a data store, in a manner appropriate to the specific data store type. When the data in one or more clients or the Data Management Service's version of data changes, the Data Management Service automatically synchronizes all client and server versions of the data. Changes to backing data stores, such as databases or directories, do not automatically propagate back into the Data Management Service and client applications. The Data Management Service feature also provides server-side and client-side conflict resolution APIs for handling data synchronization conflicts.

On the client side, you call a DataService component's `fill()` method to retrieve data from a server-side Data Management Service destination and populate an ArrayCollection. You can also call a DataService component's `getItem()` method to retrieve an individual data item. Changes to the data on the client trigger the DataService component's `commit()` method, which sends data changes to the Data Management Service destination. These changes are then propagated to other clients. The Data Management Service destination manages synchronization of data among clients and the server-side data resource. You can call the DataService component's `createItem()` or `deleteItem()` methods to add or remove data items.

The Data Management Service feature supports J2EE best practices for structuring domain models and distributing data across tiers.

Resolving data synchronization conflicts

The Data Management Service feature lets applications share and update distributed data in a synchronized manner. When more than one client changes the same piece of data at the same time, data synchronization conflicts can occur. The Data Management Service feature provides a conflict resolution API for detecting and handling data synchronization conflicts, and ensuring the integrity of distributed data by limiting further changes until conflicts are resolved.

A Data Management Service destination can detect when a data update from a client is out of date and inform the client of the conflict. The amount of server-side code required to implement conflict resolution depends on the type of adapter that you use. You can write client ActionScript code to handle conflict exceptions in the manner most appropriate for your application.

CHAPTER 51

Distributing Data in Flex Applications

51

Adobe Flex Data Services includes a client-side DataService component that you use in conjunction with the server-side Flex Data Management Service to distribute and synchronize data among multiple client applications. This topic describes how to create client-side Flex applications that can share and synchronize distributed data. For information about configuring the server-side Data Management Service, see Chapter 52, "Configuring the Data Management Service," on page 1517.

Contents

Creating a distributed data application

A Flex client application uses a client-side DataService component to receive data from, and send data to, the server-side Data Management Service. Data travels to and from Data Management Service destinations over a protocol-specific message channel. A DataService component can fill a client-side ArrayCollection object with data and manage synchronization of the ArrayCollection object's data with the versions of data in other clients and on the server. You can create DataService components in MXML or ActionScript.

A DataService component requires a valid Data Management Service destination that you configure in the services-config.xml configuration file or a file that it includes by reference. For information about Data Management Service destinations, see Chapter 52, "Configuring the Data Management Service," on page 1517.

Creating a DataService component

A DataService component manages the interaction with a server-side Data Management Service destination. You can create a DataService component in MXML or ActionScript.

The following example shows MXML code for creating a DataService component. The DataService component's destination property must reference a valid server-side Data Management Service destination.

```
<?xml version="1.0"?>
<!-- fds\datamanagement\DataServiceMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:DataService id="ds" destination="contact"/>
</mx:Application>
```

The following example shows ActionScript code for creating the same DataService component:

```
<?xml version="1.0"?>
<!-- fds\datamanagement\DataServiceAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp();">
    <mx:Script>
        <![CDATA[
            import mx.data.DataService;
            public var ds:DataService;

            public function initApp():void {
                ds = new DataService("contact");
            }
        ]]>
    </mx:Script>
</mx:Application>
```

When you create a DataService component in ActionScript, you must import the mx.data.DataService class and declare a variable of type DataService, for which you set the value to a new DataService object.

Filling and releasing data from an ArrayCollection

When you call a DataService component's `fill()` method, you fill an ArrayCollection object with the data from a Data Management Service destination. You can create an ArrayCollection object in MXML or ActionScript. The ArrayCollection API provides a set of methods and properties for manipulating a set of data; for information, see Chapter 7, "Using Data Providers and Collections," on page 161.

To release an ArrayCollection object that you filled, you call the DataService component's `releaseCollection()` method. If you call the `fill()` method again on the same ArrayCollection object with the same parameters, it fetches a fresh copy of the data. If you call the `fill()` method again with different parameters, it releases the first fill and then fetches the new one.

The first parameter of a `fill()` method is the `id` value of the ArrayCollection to fill. The values of any additional parameters depend on the type of server-side destination that you call.

For example, when you call a destination that uses the Java adapter with a custom assembler, the arguments following the `id` of the ArrayCollection could be the arguments of a corresponding server-side method that is declared in the destination. For a destination that uses the Hibernate object–relational mapping system, the arguments following the `id` of the ArrayCollection are Hibernate-specific values that depend on the Hibernate features that the destination is configured to use. The following example shows a `fill()` method for calling a Hibernate destination that uses the Hibernate Query Language (HQL):

```
myService.fill(myCollection, "flex:hql", ["from Person p where p.firstName
  = :firstName", parameterMap]);
```

For more information, see Chapter 52, "Configuring the Data Management Service," on page 1517.

The following example shows code for creating and filling an ArrayCollection object in MXML. The Application component's `creationComplete` event listener is set to the DataService component's `fill()` method, so the `contacts` ArrayCollection is filled when the application is loaded.

```
<?xml version="1.0"?>
<!-- fds\datamanagement\FillArrayCollectionMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="ds.fill(contacts);">
    <mx:DataService id="ds" destination="contact"/>
    <mx:ArrayCollection id="contacts"/>
</mx:Application>
```

The following example shows code for creating and filling an ArrayCollection in ActionScript. The DataService component's `fill()` method is called in the `initApp()` method, which is specified as the Application component's `creationComplete` event listener.

```
<?xml version="1.0"?>
<!-- fds\datamanagement\FillArrayCollectionAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp();">
    <mx:Script>
        <![CDATA[
            import mx.data.DataService;
            import mx.collections.ArrayCollection;

            public var ds:DataService;

            [Bindable]
            public var contacts:ArrayCollection;

            public function initApp():void {
                contacts = new ArrayCollection();
                ds = new DataService("contact");
                ds.fill(contacts);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Populating a data provider control with distributed data

To populate a data provider control with distributed data, you can use data binding to bind a managed ArrayCollection object to the data provider control's `dataProvider` property. When you set the associated DataService component's `autoCommit` property to `true`, changes to data in the DataGrid are automatically sent to the Data Management Service destination.

The following example shows an ArrayCollection object that is bound to a DataGrid control's `dataProvider` property:

```
<?xml version="1.0"?>
<!-- fds\datamanagement\PopulateDataGrid.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="ds.fill(contacts);">
    <mx:DataService id="ds" destination="contact"/>
    <mx:ArrayCollection id="contacts"/>
    <mx:DataGrid id="dg" dataProvider="{contacts}" editable="true">
        <mx:columns>
            <mx:DataGridColumn dataField="contactId" headerText="Id"
                editable="false"/>
            <mx:DataGridColumn dataField="firstName" headerText="First
Name"/>
           <mx:DataGridColumn dataField="lastName" headerText="Last Name"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Sending changes from a managed ArrayCollection object

By default, a DataService component's `commit()` method is automatically called when data changes in the ArrayCollection object that it manages. You can also call the `commit()` method manually and set a DataService component's `autoCommit` property to `false` to allow only manual calls to the `commit()` method.

The following example shows a manual update operation on an item in an ArrayCollection object that is managed by DataService component:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">
  <mx:Script>
    <![CDATA[
      import mx.data.DataService;
      import mx.collections.ArrayCollection;
      import samples.customer.Customer;
      ...
      public function initApp():void {
                var customers:ArrayCollection = new ArrayCollection();
                var customerService:DataService = new
                  DataService("customers");
                customerService.autoCommit = false;
                var customer:Customer = customers.getItemAt(4);
                customer.name = "CyberTech Enterprises";
                customerService.commit();
           }
           ...
    ]]>
```

```
  </mx:Script>
</mx:Application>
```

The DataService component creates a single update message that includes the change to the `customer.name` property. This message is sent to the destination when the DataService component's `commit()` method is called.

The following example shows two manual update operations on an item in an ArrayCollection object that is managed by a DataService component:

```
...
var customer:Customer = customers.getItemAt(4);
var oldName:String = customer.name;
customer.name = "CyberTech Enterprises";
customer.name = oldName;
customerService.commit();
...
```

The DataService component attempts to log two updates that cancel out each other. When the value of the `customer.name` property is changed to `"CyberTech Enterprises"`, the DataService component creates a single update message. On the subsequent change back to the old name by using `customer.name = oldName`, the original update message for the customer's name is removed. The result is that nothing is sent when the `commit()` method is called.

The following example shows the addition and removal of a customer to an ArrayCollection object that is managed by DataService component.

```
<?xml version="1.0"?>
<!-- fds\datamanagement\AddRemoveItem.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.data.DataService;
            import samples.customer.Customer;
            public function initApp():void {
                var customers:ArrayCollection = new ArrayCollection();
                var customerService:DataService =
                  new DataService("customers");
                customerService.autoCommit = false;
                customers.addItemAt(new Customer(), 4);
                // Remove the previously added customer.
                customers.removeItemAt(4);
                customerService.commit();
            }
        ]]>
    </mx:Script>
</mx:Application>
```

The DataService component attempts to log two updates that cancel out each other. When the `addItemAt()` method is called, the DataService component creates a create message for the `customers` destination. On the subsequent call to `removeItemAt(4)`, the previous create message is removed. Nothing is sent when the `commit()` method is called.

NOTE Calling an ArrayCollection object's `setItemAt(x)` method is equivalent to calling the `removeItemAt(x)` method and then calling the `addItemAt(..., x)` method.

In all of the previous examples, the DataService component uses only the identity of the item specified to determine if a new update cancels pending updates. The following example also shows this type of behavior:

```
...
var customer:Customer = new Customer("1099");
customer.name = "CyberTech Enterprises";
// At location 4 in the list, a customer object with identity 1099 exists.
customers.setItemAt(customer, 4);
customerService.commit();
...
```

In this example, the DataService component does not send any changes to the server even though a new Customer object is created and the value of its `name` property is changed.

Working with single data items

The mx.data.DataService class has several methods for working with individual data items. The following table describes these methods:

Method	Description
`createItem()`	Lets you create a new data item without working with an ArrayCollection. An example of when this method is useful is a call center application where a customer fills in a form to create a single ticket item. In contrast, call center employees would have to see all the ticket items, so their application would fill an ArrayCollection with data items.
`getItem()`	Lets you get a single data item from the item's identity. An example of when this method is useful is in a master-detail relationship, such as the relationship between an order and order items. When you click on one of the order items in a DataGrid, you want to go back out to another destination and get a complete item that includes several more properties from another destination. This would let you bring a minimal amount of data for the order into the initial ArrayCollection and only get complete items on an individual basis when needed.

Method	Description
`deleteItem()`	Deletes an item which is managed using a `createItem()`, `getItem()`, or `fill()` method call. The delete is sent to the server as soon as the transaction is committed.
`releaseItem()`	Releases an item from management. This method should be called for each `getItem()` or `createItem()` call that you are finished with to prevent a memory leak on the client (and the server if `autoSyncEnabled` is set to `true`).

The following example shows a method that gets a specific data item when a DataGrid control changes. You get the item from the `ResultEvent.result` event. The identity value sent to the destination is `companyId`, which is set to the `companyId` of the currently selected item in the DataGrid control. The destination retrieves items based on their identity, which is specified in the destination definition in the configuration file.

```
  <mx:Script>
    <![CDATA[
...
       private function companyChange() {
         dsCompany.getItem({companyId: dg.selectedItem.companyId});
       }
    ]]>
  </mx:Script>
...
```

Connecting and disconnecting

A DataService component starts out disconnected from its server-side destination. The first time you perform an operation, the DataService component tries to connect to the destination. If the operation succeeds, a result event is sent to the DataService component. If the operation fails, a fault event is sent. You can call the `DataService.disconnect()` method to force a connected client to disconnect; in this case, the DataService component keeps a copy of its managed data and automatically resubscribes to pick up changes when you reconnect.

You can also call `DataService.release()` method to release all managed objects fetched by a DataService component.

About the DataStore object

The Data Management Service keeps outgoing and incoming data changes in an object called DataStore, which can be shared by more than one DataService component. By default, two DataService components share the same DataStore object when there are associations between them or you manually set one DataService component's `dataStore` property to refer to the DataStore object of another DataService component.

When you call a DataService component's `commit()` method, it commits all of the changes for the DataStore object, which includes the changes for all DataService components that share that DataStore object. Similarly, when you change one DataService component's `autoCommit` property, the value is changed for all DataServices components that share the same DataStore object.

Handling errors

The DataService component dispatches fault events for errors that occur when processing an operation. This includes errors that occur when connecting to the server, as well as errors that are sent by an assembler class in response to processing an operation.

For more information, see the documentation for the mx.data.errors package in the *Adobe Flex 2 Language Reference*.

A DataService component automatically connects to a ChannelSet when it performs its first operation. When errors occur in your clients when processing a `commit()` request, the changes in the commit are put back into the uncommitted changes. You can choose to revert those changes by calling the DataService component's `revertChanges()` method with an item argument, or you can the `revertChanges()` method with no argument to revert all changes.

Controlling whether clients receive pushed changes

By default, the Data Management Service detects data changes made from Flex clients and the server push API and propagates those changes to other clients. You can change this default behavior for a specific destination by setting `auto-sync-enabled` to `false` in the destination configuration.

To turn on the auto-sync behavior for a specific ArrayCollection on the client side, set the DataService component's `autoSyncEnabled` property to `true` before you call its `fill()` method. Similarly, to retrieve a managed item from an individual reference, you set the DataService component's `autoSyncEnabled` property to `true` before you call its `getItem()` or `createItem()` method.

Changing the value of the `autoSyncEnabled` property does not affect existing managed objects on that client. It only affects `fill()`, `getItem()` and `createItem()` method calls you make after changing the value. This lets you to have managed instances with the `autoSyncEnabled` property set to `true` and others with the `autoSyncEnabled` property set to `false` from the same destination in the same client.

Controlling pushed changes

By default, when a client detects changes, the changes are immediately applied to the Data Management Service. You can turn this off for a given destination by setting a DataService component's `autoMerge` property to `false`. When pending changes come in, the `mergeRequired` property is set to `true` on the `DataStore` object. To merge the pending changes, you call the `DataService.dataStore.merge()` method. To avoid conflicts, you should merge changes before you modify any data locally on the client.

NOTE

A client does not make page requests while there are unmerged updates if a DataService component's `autoMerge` property is set to `false`, paging is enabled, and the page size is a non-zero value. You can test for this case by checking the value of the `mergeRequired` property of the DataService component or its associated DataStore object.

Providing authorization information

The DataService component's `setCredentials()` method lets you provide authorization information required by server-side security constraints. The `logout()` method removes those credentials. For more information on providing authorization information, see "Securing destinations" on page 1379.

Mapping client-side objects to Java objects

To represent a server-side Java object in a client application, you use the `[RemoteClass(alias=" ")]` metadata tag to create an ActionScript object that maps directly to the Java object. You specify the fully qualified class name of the Java class as the value of `alias`. This is the same technique that you use to map to Java objects when using RemoteObject components.

You can use the `[RemoteClass]` metadata tag without an alias if you do not map to a Java object on the server, but you do send back your object type from the server. Your ActionScript object is serialized to a Map object when it is sent to the server, but the object returned from the server to the clients is your original ActionScript type.

To create a managed association between client-side and server-side objects, you also use the `[Managed]` metadata tag or explicitly implement the mx.data.IManaged interface.

The CRM application that is included in the Flex Data Services sample applications provides a good example of a managed association. T following example shows the source code for the client-side ActionScript Company class. which has a managed association with the server-side Java Company class:

```
package samples.crm
{
import mx.collections.ArrayCollection;
  [Managed]
  [RemoteClass(alias="samples.crm.Company")]
  public class Company
  {
    public var companyId:int;

    public var name:String = "";

    public var address:String = "";

    public var city:String = "";

    public var state:String = "";

    public var zip:String = "";

        public var industry:String = "";

    public function Company()
    {
    }

  }
}
```

The following example shows the source code for the corresponding server-side Java Company class:

```
package samples.crm;

import java.util.Set;

public class Company
{
```

```
private int companyId;

private String name;

private String address;

private String city;

private String zip;

private String state;

  private String industry;

public String getAddress()
{
  return address;
}

public void setAddress(String address)
{
  this.address = address;
}

public String getCity()
{
  return city;
}

public void setCity(String city)
{
  this.city = city;
}

public int getCompanyId()
{
  return companyId;
}

public void setCompanyId(int companyId)
{
  this.companyId = companyId;
}

public String getName()
{
  return name;
}

public void setName(String name)
```

```
  {
    this.name = name;
  }

  public String getState()
  {
    return state;
  }

  public void setState(String state)
  {
    this.state = state;
  }

  public String getZip()
  {
    return zip;
  }

  public void setZip(String zip)
  {
    this.zip = zip;
  }

    public String getIndustry()
    {
        return this.industry;
    }

    public void setIndustry(String industry)
    {
        this.industry = industry;
    }

    public String toString()
    {
        return "Company(companyId=" + companyId + ", name=" + name + ",
  address=" + address +
                       ", state" + state + ", zip=" + zip + " industry=" +
  industry + ")";
    }
}
```

Handling data synchronization conflicts

When working with distributed data applications, there are often times when clients try to make data changes that conflict with changes that were already made. The following scenarios are core cases where data synchronization conflicts occur:

Conflict type	Description
Update with stale data	The server detects a conflict when a client commits changes because the data has changed since the client received the data.
Client is pushed changes when it has uncommitted conflicting changes	The client is in the midst of changing the data which conflicts with changes pushed to it from the server.
Delete with stale data	Server detects a conflict when the client tries to delete an item with a stale version of that object.
Client is pushed changes to an item is has deleted locally	A client has an uncommitted request to delete data when an update has just occurred to the same data, and a data message has been pushed to the client from the server.
Update to deleted data without client push	A client tries to update data that has already been deleted from the server.
Update to deleted date with client push	A client tries to update data that has already been deleted, and a data message has been pushed to the client from server. The client tries to send the update, but there is a conflict on client.

The following example shows a simple event handler that displays an Alert box on the client where a conflict occurs, and accepts the server version of the data:

```
...
public function useDS:void {
...

  ds.addEventListener(DataConflictEvent.CONFLICT, conflictHandler);
...
}

public function conflictHandler(event:DataConflictEvent):void {
  var conflicts:Conflicts = ds.conflicts;
  var c:Conflict;
  for (var i:int=0; i<conflicts.length; i++) {
    c = Conflict(conflicts.getItemAt(i));
    Alert.show("Reverting to server value", "Conflict");
    c.acceptServer();
```

```
    }
}
```

For more information about the mx.data.Conflicts and mx.data.Conflict, and mx.data.events.DataConflictEvent classes, see the *Adobe Flex 2 Language Reference.*

When you use the Java adapter in your destination, you write a sync method in your assembler class that handles data synchronization and throws a DataSyncException if the previous version of the data does not match what is currently in the data resource. A DataSyncException results in a DataConflictEvent on the client. For more information about sync methods, see "Using the fill-method and sync-method approach" on page 1533.

You can also let the user decide which version of data gets used when a conflict occurs, as the following code snippets show. PersonForm represents a custom component that displays all of the properties of the Person object assigned to its `dataSource` property.

```
// Conflict event handler:
function resolveConflictsHandler():void {
displayConflictsScreen();
}
...
<!-- Conflicts screen MXML code: -->
<PersonForm id="serverValue" editable="false"
  dataSource="{ds.conflicts.current.serverObject}"/>

<PersonForm id="clientValue"
  dataSource="{ds.conflicts.current.clientObject}"/>

<PersonForm id="originalValue"
  dataSource="{ds.conflicts.current.originalObject}" editable="false"/>

<mx:Button label="Accept Server" click="ds.conflicts.current.acceptServer"
  />
<mx:Button label="Accept Client" click="ds.conflicts.current.acceptClient"
  />
...
```

CHAPTER 52

Configuring the Data Management Service

52

To enable distributed data in an Adobe Flex client application, you connect to the Flex Data Management Service by declaring a connection to a server-side Data Management Service destination in an `<mx:DataService>` tag or the corresponding ActionScript API. You configure destinations as part of the Data Management Service definition in the services-config.xml file or a file that the services-config.xml file includes by reference. This topic describes how to configure destinations and the data adapters they use as well as how to push data from the server to clients.

For information about using the client-side DataService component and connecting to a Data Management Service destination in MXML or ActionScript, see Chapter 51, "Distributing Data in Flex Applications," on page 1501.

Contents

About Data Management Service configuration

The most common tasks that you perform when configuring the Data Management Service are defining Data Management Service destinations, applying security to destinations, and modifying logging settings. A *Data Management Service destination* is the endpoint that you send data to and receive data from when you use the Data Management Service to provide data distribution and synchronization in your applications. You configure Data Management Service destinations in the data service section of the services-config.xml file or a file which that file includes by reference. By default, the services-config.xml file is located in the WEB_INF/flex directory of the web application that contains Adobe Flex Data Services. In the services-config.xml files that ship with Flex Data Services, Adobe references individual configuration files for the Data Management Service, Message Service, Remoting Service, and Proxy Service.

The following example shows a basic Data Management Service configuration. It contains one destination that uses the Java adapter to interact with a data resource. The ActionScript adapter is configured as the default adapter.

```
<service id="data-service" class="flex.data.DataService"
  messageTypes="flex.data.messages.DataMessage">

  <adapters>
    <adapter-definition id="actionscript"
      class="flex.data.adapters.ASObjectAdapter" default="true"/>
    <adapter-definition id="java-adapter"
      class="flex.data.adapters.JavaAdapter"/>
  </adapters>

  <default-channels>
    <channel ref="my-rtmp"/>
  </default-channels>

  <destination id="contact">
    <adapter ref="java-adapter" />
      <properties>
        <source>dev.contacts.ContactAssembler</source>
        <scope>application</scope>
        <cache-items>true</cache-items>
        <metadata>
          <identity property="contactId"/>
        </metadata>

        <network>
          <session-timeout>20</session-timeout>
          <paging enabled="true" pageSize="10" />
```

```
            <throttle-inbound policy="ERROR" max-frequency="500"/>
            <throttle-outbound policy="REPLACE" max-frequency="500"/>
          </network>

          <server>
            <fill-method>
              <name>loadContacts</name>
            </fill-method>
            <fill-method>
              <name>loadContacts</name>
              <params>java.lang.String</params>
            </fill-method>
            <sync-method>
              <name>syncContacts</name>
            </sync-method>

          </server>
        </properties>
    </destination>
</service>
```

Data Management Service destinations

A Data Management Service destination references one or more message channels that transport messages, and contains network- and server-related properties. It can also reference a *data adapter*, which is server-side code that lets the destination work with data through a particular type of interface, such as a Java object. You can also configure a specific data adapter as the default data adapter, in which case you do not need to reference it inside the destination. A destination can also reference or define security constraints for a destination.

To use the Java adapter you write an assembler class, which is unlike what you do for other adapters. An *assembler class* is a Java class that interacts indirectly or directly with a data resource. A common design pattern is for the assembler to call a data access object (DAO) that calls a data resource.

Message channels

A Data Management Service destination references one or more message channels, which are defined elsewhere in the configuration files. The two most common channels that you use for publish-subscribe messaging are the Realtime Message Protocol (RTMP) channel, and the Action Message Format (AMF) channel with message polling enabled.

The Realtime Message Protocol (RTMP) channel maintains a connection between the client and the server, so the client does not have to poll the server. The Action Message Format (AMF) channel with polling enabled polls the server for new messages. For more information about messaging channels, see "Configuring message channels" on page 1360.

Data Management Service data adapters

A data adapter provides the infrastructure for interacting with data resources. You specify data adapters in a destination. You can also specify adapter-specific settings.

For more information about data adapters, see "Configuring Data Management Service destinations" on page 1520.

Security

You can secure a destination by using a *security constraint*, which defines the access privileges for the destination. You use a security constraint to authenticate and authorize users before letting them access a destination. You can specify whether to use basic or custom authentication, and indicate the roles required for authorization.

You can declare a security constraint inline in a destination, or you can declare it globally and reference it by its `id` value in a destination.

For more information about security, see "Securing destinations" on page 1379.

Configuring Data Management Service destinations

Some aspects of configuring Data Management Service destinations are general for all Data Management Service destinations, and others are determined by the particular data adapter that a destination uses. This section describes general configuration for Data Management Service destinations.

Referencing message channels

Messages are transported to and from a destination over a message channel. A destination references one or more message channels that are defined in the services-config.xml file. The destination uses the first of the referenced channels that is available. If a channel cannot be reached, the destination uses the next available channel. You can also configure default message channels; if you do this, you do not have to reference any channels in the destination.

The following example shows a destination's channel references. The `samples-rtmp` channel is listed first, so the destination attempts to use it first.

```
...
  <destination id="contact">
...
    <channels>
      <channel ref="samples-rtmp"/>
      <channel ref="samples-amf-polling"/>
    </channels>
...
  </destination>
...
```

For more information about message channels, see "Configuring message channels" on page 1360.

Configuring network elements

The `properties` section of a destination definition contains a `network` section for defining client-server messaging behavior. The following example shows the `network` section of a destination definition:

```
...
  <destination id="contact">
    <properties>
...
        <network>
          <session-timeout>20</session-timeout>
          <paging enabled="true" pageSize="10"/>
          <throttle-inbound policy="ERROR" max-frequency="500"/>
          <throttle-outbound policy="REPLACE" max-frequency="500"/>
          <!--<cluster ref="default-cluster"/>-->
        </network>
  ...
    </properties>
  </destination>
...
```

Data Management Service destination definitions use the following network-related elements:

Element	Description
`cluster`	The `ref` attribute of the `cluster` element references the name of a software cluster, which is configured in the clusters section of the services-config.xml file. For information about clusters, see "Using software clustering" on page 1388.
`paging`	The `enabled` attribute of the `paging` element indicates whether data paging is enabled for the destination. When you enable paging, the `pageSize` attribute indicates the number of records to be sent to the client when the client-side `DataService.fill()` method is called. This element is optional; the default value is `false`.
`session-timeout`	(Optional) The `session-timeout` element specifies the idle time in minutes before a subscriber is unsubscribed. When you set the value to `0` (zero), subscribers are not forced to unsubscribe automatically. The default value is 20.
`throttle-inbound`	The `max-frequency` attribute of the `throttle-inbound` element controls how many messages the server can receive per second. The `policy` attribute indicates what to do when the message limit is reached. The `policy` attribute can have the following values: • `ERROR` indicates that an error should be returned if the limit is reached. • `IGNORE` indicates that no error should be returned if the limit is reached.
`throttle-outbound`	The `max-frequency` attribute of the `throttle-outbound` element controls how many messages the server can send per second. The `policy` attribute indicates what to do when the message limit is reached. The `policy` attribute can have the following values: • `ERROR` indicates that an error should be returned if the limit is reached. • `IGNORE` indicates that no error should be returned if the limit is reached. • `REPLACE` indicates that the previous message should be replaced when the limit is reached.

Using transactions

By default, Flex encapsulates client-committed operations in a single J2EE distributed transaction. All client data changes are handled as a unit. If one value cannot be changed, changes to all other values are rolled back. Distributed transactions are supported as long as your Java 2 Enterprise Edition (J2EE) server provides support for J2EE transactions (JTA).

The boldface text in the following example shows the `use-transactions` property set to `true`, which is the default value:

```
...
  <destination id="contact">
      <properties>
        <use-transactions>true</use-transactions>
    </properties>
  </destination>
...
```

Uniquely identifying data items

A Data Management Service destination contains one or more `identity` elements that you can use to designate data properties to be used to guarantee unique identity among items in a collection of data.

The boldface text in the following example shows an identity property:

```
...
  <destination id="contact">
    <properties>
...
    <metadata>
      <identity property="name"/>
    </metadata>
  </properties>
  </destination>
...
```

The `identity` element takes an optional `type` attribute, which is the Java class of the specified identity property. You must use this when your identity type may not have a one-to-one correspondence with ActionScript types. The most common problem is that an ActionScript Number is converted to either a long or an integer, depending on its value. The code in the following example shows an `identity` element with a `type` attribute:

```
<identity property="id" type="java.lang.Long">
```

A common mistake is to try to use one `identity` element for all properties, as the following example shows:

```
<metadata>
  <!— Don’t do this. -->
  <identity property="firstName,lastname"/>
</metadata>
```

Instead, you can specify multiple identity elements for a multiple field identity, as the following example shows:

```
<metadata>
  <identity property="firstName"/>
  <identity property="lastName"/>
</metadata>
```

Caching data items

By default, the Data Management Service caches items returned from `fill()` and `getItem()` calls and uses cached items to implement paging and to build object graphs when implementing lazy associations. This causes a complete copy of the managed state of all active clients to be kept in each server's memory. You can turn off this caching of memory by setting `cache-items` to `false`, as the following example shows:

```
...
  <destination id="contact">
    <properties>
...
      <cache-items>false</cache-items>
...
  </properties>
  </destination>
...
```

If you set `cache-items` to `false`, you must specify a method to support paging or lazy associations in a `get-method` element when using this destination.

Synchronizing data automatically

The `auto-sync-enabled` element controls the default value of the client-side `DataService.autoSyncEnabled` property for clients that are using a destination. The default value of this element is `true`.

The following example shows an `auto-sync-enabled` element:

```
...
  <destination id="contact">
    <properties>
...
      <auto-sync-enabled>false</auto-sync-enabled>
...
  </properties>
  </destination>
...
```

Working with data adapters

The Data Management Service creates and updates data messages based on operations performed by Flex client applications on ArrayCollection objects and single managed objects, and on persistent data in an enterprise middle tier or resource tier. The process of accessing data resources occurs through the Data Management Service's contract with a data adapter. A *data adapter* is responsible for updating the persistent data store in a manner appropriate to the specific data store type; for the Java adapter, the data adapter is responsible for delegating this update responsibility to the assembler class.

The Java adapter and the ActionScript object adapter are included with Flex Data Services.

The Java adapter passes data changes to methods available on a Java class, called the Java assembler. This adapter lets Java developers employ the Data Transfer Object (DTO) design pattern. The Hibernate assembler is a specialized assembler that provides a bridge from the Data Management Service to the Hibernate object-relational mapping system; for more information, see "Using a Hibernate destination" on page 1547.

You use the ActionScript object adapter when a Flex client application is the only creator and consumer of transient data objects and there is no back-end data resource. The Data Management Service uses the ActionScript object adapter to manage objects in memory.

The following sections describe these adapters and how to use them.

Using the Java adapter

The Java adapter lets you synchronize Java objects on the server with ActionScript objects on the client. This adapter passes data changes to methods available on a Java class called an assembler. You use the Java adapter for custom assembler classes and for the Hibernate assembler, which lets you work with the Hibernate object-relational mapping system.

For an application that doesn't use Hibernate, you first decide on the destinations required to support your application objects, and then you define an assembler class for each destination. Your assembler class must only support one or more fill methods. These methods implement queries, each of which returns a Collection of objects to the client code. If your client must update objects, then you need a sync method or individual create, update, and delete methods.

Depending on your server architecture, you could choose to directly transfer your domain model objects to the client, or you could implement a distinct layer of data transfer objects (DAOs) that replicate only the model state you want to expose to your client tier.

Understanding assemblers

The Data Management Service supports two approaches to writing Java assembler classes. One is designed for simple applications and lets you support just the Data Management Service features that you require. In this approach, you declare individual methods by using `fill-method` and `sync-method` XML elements in a destination definition to map to methods of your assembler class. You must only add the methods required for the feature set required for your clients. For example, if your client application is read-only, you may only have to provide a get method and or a fill-method as needed by your clients. A get method is required on an assembler if a client uses the `DataService.getItem()` method. It is also required if you set `cache-items` to `false` and have lazy associations that point to this item, or you are using paging.

Later, if you want your assembler to support updates, you can add a `sync-method` definition. In this approach, your class does not have to implement any special Flex interfaces, so in some simple cases you may be able to use this without writing Java code to expose functionality to Flex applications.

The other approach, referred to as the Assembler interface approach, is a more traditional way to interface with Java components. You extend the flex.data.assemblers.AbstractAssembler, which implements the flex.data.assemblers.Assembler interface, and you override methods such as the `getItem()`, `fill()`, `createItem()`, `updateItem()` methods. The AbstractAssembler class provides default implementations of all of the Assembler interface methods, so this approach provides some of the benefits of the first approach. The AbstractAssembler class and Assembler interface are documented in the public Flex Data Services Javadoc documentation.

You can also combine both approaches in the same assembler class. If you have an XML definition for a method, the Data Management Service calls that method instead of the equivalent method in the Assembler interface. Adding fill and count methods by using XML declarations in a destination definition can be a useful way to expose queries to your clients. It prevents your assembler's `fill()` method from having to do the dispatch for different types of fill methods. If a matching `fill-method` element is found in the destination, that method is called. Otherwise, the Data Management Service calls the assembler's implementation of the `fill()` method defined in the Assembler interface.

Object types and relationships

The Java adapter supports flat objects that only contain primitive fields and properties, as well as objects that have complex objects as properties. If you have complex objects as properties and those objects have their own identity property or properties, you can choose to define a separate destination to manage the synchronization of those instances based on the identity property or properties specified in `identity` element in the destination definition. If there is no identity property for the complex object properties and each instance is owned by its parent object, you can use the parent destination to manage the state of the object properties.

An example of these different ways of managing object synchronization is a Customer object that has a nested Address object. If that Address is not used by other Customer instances, the Customer destination can be responsible for persisting the state of the Address instances as they change. If the Address has its own identity and can be shared by multiple customers, it would make more sense to create a separate Address destination to persist Address instances. In this case, the Customer destination would only be responsible for persisting the identity or identities of the addresses that Customer refers to. When you use a property to represent a relationship to objects managed by another destination, it is a called a managed association property. For more information on this topic, see "Managing hierarchical collections" on page 1556.

In some situations, the managed associations approach is not the best way to handle the relationship between objects. As an alternative, you can implement a relationship between two objects by using queries where one of the parameters of the `fill()` method defined in the assembler is the identity of the related object. This Assembler interface approach is more efficient for large collections of objects than the managed association approach. For example, suppose you have a Company object that has a large number of Employee instances. If you implement this by using a managed association property (for example, company.employees), the entire list of employees is sent from the server to the client when data is returned for the fill request, and from the client to the server when an update is made. For more information, see "Implementing an object relationship in a fill method" on page 1567.

When you use the managed association approach, even with the lazy loading feature, the identities of all of the referenced employees would be sent. For information about lazy loading, see "Implementing a managed association" on page 1559.

If a company has 10,000 employees, that could be computationally expensive. Instead, you could have a fill method on the employee destination that takes the company `id` as a parameter that would return the employees that exist for that company. With this approach, you can use the Data Management Service paging feature to send the first page of employees. When a new employee is added or removed from a company, you would have to send only the employee's data. The unaffected employees would not be sent from the client to the server.

NOTE

When you compile code that uses Flex Data Services Java APIs, you must include the messaging.jar and flex-messaging-common.jar files in your classpath.
The compiled assembler class and any other required classes, such as a DAO class, must be available in the web application's classpath.

Using strong and anonymous types with the Java adapter

The properties of a strongly typed object are defined at compile time by using declared fields or get and set methods. An anonymous type is usually represented in Java by a java.util.Map, and in ActionScript by an instance of type Object. For anonymous objects, type checking is performed at run time, not compile time.

You can use either strongly typed objects or anonymous objects with Data Management Service on either the client or the server. For example, to use anonymous objects on both the client and server, your assembler returns java.util.Map instances that are converted to Object instances on the client. When the client modifies those Object instances, they are sent back to the assembler as java.util.Map instances.

To use strongly typed objects on the client and server, you define an ActionScript class that has explicit public properties for the data you want to manage with the `[Managed]` metadata tag. You map this class to a Java class on the server by using the `[RemoteClass(alias="`*`java-class-name`*`")]` metadata tag. Your Data Management Service destination in this case does not have to be configured with either class explicitly, which allows a single destination to support whatever instances your client sends to it. The client just has to expect the instances returned by the Java assembler, and the Java assembler has to recognize instances returned by the client. This allows one destination to support an entire class hierarchy of instances and you don't have to configure the Data Management Service destination explicitly for each class.

The Data Management Service also supports a technique that allows your Java assembler to return strongly typed Java instances that are converted to anonymous types on the client. In this case, you do not have to maintain ActionScript classes for each of your Java classes. When those anonymous instances are modified, the server receives them as java.util.Map instances. Data Management Service supports the `item-class` element in the `properties` section of a destination definition to automatically convert these instances back into a single strongly typed Java class that the Java assembler expects. Set the item-class element to refer to the Java class that your assembler expects. This pattern only supports the common case where each destination only returns instances of a single Java class. If your assembler must operate on a class hierarchy and you do not want to use strongly typed instances on the client, you must convert from the java.util.Map to your strongly typed instances in your assembler yourself.

You can also write a Java assembler that uses anonymous objects on the server that are serialized to strongly typed ActionScript classes on the client. To do this, you create an instance of the class flex.messaging.io.amf.ASObject that implements the java.util.Map interface. Set the type property of that instance to be the fully qualified class name of the ActionScript class that you want that instance to serialize to on the client, and store the property values that ActionScript instance expects as key-value pairs in the java.util.Map. When that object is received by the client, the serialization code creates an instance of the proper type.

Configuring a destination that uses the Java adapter

As previously described, when you work with the Java adapter, you have the following choices for how you write your assemblers:

- Use `fill-method` and `sync-method` XML elements in the destination to map to methods on an arbitrary class.
- Write an assembler class that implements the flex.data.assemblers.Assembler interface and provides a `fill()` method implementation.
- Use a combination of the first two approaches. When you combine the two approaches, if the signature of a fill request from a client doesn't match a `fill-method` declaration, the Data Management Service calls the `fill()` method defined in the assembler.

The choice that you make determines which XML elements you include in your destination and the Java code you write. For general information, see "Understanding assemblers" on page 1526.

The following example shows a Data Management Service destination that uses a Java assembler and contains `fill-method` and `sync-method` elements.

```
<service id="data-service" class="flex.data.DataService"
  messageTypes="flex.data.messages.DataMessage">
...
  <adapters>
    <adapter-definition id="java-adapter"
      class="flex.data.adapters.JavaAdapter"/>
  </adapters>
...
  <destination id="contact">
    <adapter ref="java-adapter"/>
    <properties>
      <metadata>
        <identity property="contactId"/>
      </metadata>

      <source>samples.contact.ContactAssembler</source>
      <scope>application</scope>

      <network>
        <paging enabled="true" pageSize="10"/>
        <throttle-inbound policy="ERROR" max-frequency="500"/>
        <throttle-outbound policy="REPLACE" max-frequency="500"/>
      </network>

      <server>

        <fill-method>
          <name>loadContacts</name>
        </fill-method>

        <fill-method>
          <name>loadContacts</name>
          <params>java.lang.String</params>
        </fill-method>

        <sync-method>
          <name>syncContacts</name>
        </sync-method>

        <get-method>
          <name>getPerson</name>
        </get-method>

        <count-method>
          <name>getCount</name>
        </count-method>
      </server>
```

```
    </properties>
  </destination>
</service>
```

The `source` element specifies the assembler class, which is the class that handles data changes. The `scope` element specifies the scope of the assembler; the valid values are `application`, `session`, and `request`.

You use `fill-method` and `sync-method` elements to map arbitrary methods on an assembler as fill and sync methods for filling and synchronizing data in client-side ArrayCollection objects. The `fill-method` elements specify the assembler methods that are used to fill client-side ArrayCollection objects with data. The `sync-method` element specifies the method used to synchronize data on clients and the server-side data resource. You can also use the `get-method` element to specify the method used to get single data items, and the `count-method` element to specify the method used to get a count of all data items in the data resource.

Detecting changes to fill method results

You use fill methods to implement queries that return a collection of items for a particular destination. When a client executes a fill method with its `DataService.autoSyncEnabled` property set to `true`, it listens for changes to the results of this query so that it can update the list displayed on the client. By default, the Data Management Service implements this by using an exhaustive technique called auto-refresh, which usually ensures that the client's results stay in sync with the database.When you use this technique, whenever an item is created or updated either by the client or the server push API, the Data Management Service re-executes any `fill()` methods currently managed on active clients. It compares the IDs of the returned items to its cached results; if any have been added or removed, it builds an update collection message that is sent to the clients that have that fill method cached. For simple applications, this may be all that you need; however, when the number of fill methods with unique parameters managed by clients grows large, the auto-refresh technique can result in the execution of many unnecessary queries.

There are two techniques for keeping fill methods from being executed unnecessarily. You can disable auto-refresh, and use the `refreshFill()` method of the flex.data.DataServiceTransaction class (the server push API) to explicitly invalidate a fill or set of fill methods by specifying a pattern that matches the parameters of the fill methods you want to refresh. Alternatively, you can leave auto-refresh enabled, and set it so that it only re-executes the fill methods whose results are changed by a specified item.

In the second approach, you supply a method that gets called for each item that is created or updated for each unique set of fill parameters currently managed on active clients. This method returns a value that indicates how the fill should be treated for that change.

How this method works depends on whether your assembler uses XML-based `fill-method` and `sync-method` elements in a destination or implements the Assembler interface. For more information about the `fill-method` and `sync-method` approach, see "Using the fill-method and sync-method approach" on page 1533. For more information about the Assembler interface approach, see "Using the Assembler interface approach" on page 1538

When you use `fill-method` elements, you define a `fill-contains-method` element that points to a method in your Assembler class. That method takes the changed or created item and the List of fill parameters and returns `true` or `false` depending on whether the item is in that fill. If the fill is not ordered and the method returns `true`, the item is added to the end of the list of managed items for that fill. If the fill is ordered and you return `true`, the fill method is re-executed to get the proper ordering of the list. If the method returns `false`, the fill method is left as is for that update.

The following example shows `fill-method` elements with `fill-contains-method` elements for unordered and ordered fills:

```
<fill-method>
  <name>loadUnorderedPatternGroups</name>
  <params>java.lang.String,java.lang.Boolean</params>
  <ordered>false</ordered>
  <fill-contains-method>
    <name>unorderedPatternContains</name>
  </fill-contains-method>
</fill-method>
<fill-method>
  <name>loadOrderedPatternGroups</name>
  <params>java.lang.Boolean,java.lang.String</params>
  <ordered>true</ordered>
  <fill-contains-method>
    <name>orderedPatternContains</name>
  </fill-contains-method>
</fill-method>
```

The following example shows the `unorderedPatternContains()` and `orderedPatternContains()` methods in the corresponding assembler:

```
...
  public boolean unorderedPatternContains(Object[] params, Object group) {
    AssocGroup g = (AssocGroup)group;
    if (g.getName().indexOf((String)params[0]) != -1)
    return true;
    return false;
}

  public boolean orderedPatternContains(Object[] params, Object group) {
    AssocGroup g = (AssocGroup)group;
    if (g.getName().indexOf((String)params[1]) != -1)
    return true;
    return false;
}
```

When you implement the Assembler interface, a `refreshFill()` method is called for each active fill managed on the client. It is given the list of fill parameters and the changed item as well, but the `refreshFill()` method returns one of three codes that indicate whether the fill should be re-executed, the item is appended to the fill, or the fill should be left unchanged for this operation.

For code examples of both techniques for keeping fill methods from being executed unnecessarily when you implement the Assembler interface, see "Using the Assembler interface approach" on page 1538.

Using the fill-method and sync-method approach

In the `fill-method` and `sync-method` elements of a destination that uses the Java adapter, you can declare methods that are called when data operations occur on a Flex client application. You use these elements to map arbitrary methods of an assembler as fill and sync methods. The method names that you specify in these elements correspond to methods in the assembler class. You never use a sync method if the items are read-only.

If the signature of the fill request sent for a client does not match a `fill-method` element, the Data Management Service calls the `fill()` method defined in the assembler if it implements the `flex.data.assemblers.Assembler` interface.

When you use `fill-method` elements, each fill method is identified by the types of parameters it is declared to receive in the XML file. You must ensure that you do not declare two fill methods that take the same types of arguments or an error occurs when you attempt to invoke the fill method. Another important consideration is that a null value passed to a fill method acts like a wildcard that matches any type. If you expect to have many fill methods, you may want to have them take a name or ID as the first parameter to uniquely identify the query you want to perform. You can then do the dispatching for queries with different names within your fill method.

You can implement any number of fill methods to fill a client-side ArrayCollection with data items. You specify fill methods in the `<fill-method>` section of the destination; each of these methods is assigned as a fill method that can accept an arbitrary number of parameters of varying types and returns a List object. The types must be available at run time. Any call to a client-side DataService component's `fill()` method results in a server-side call to the appropriate Java fill method with the parameters provided by the client.

In a `sync-method` element, you specify a method that accepts a change list as its only input parameter. You can choose to implement the sync method as a single method that takes a list of ChangeObjects, or you can extend the AbstractAssembler class and override the `updateItem()`, `createItem()`, and `deleteItem()` methods.

The change list is a java.util.List implementation that contains objects of type flex.data.ChangeObject. Each ChangeObject in the list contains methods to access an application-specific changed Object instance, and convenience methods for getting more information about the type of change that occurred and for accessing the changed data members. You should only use the iterator on this list to iterate through the ChangeObjects. Use of the get method on the list is not allowed.

The flex.data.ChangeObject class is included in the public Flex Data Services Javadoc documentation.

The following example shows an assembler class that calls a DAO to work with a SQL database. The two `loadContacts()` methods are the fill methods, and the `syncContacts()` method is the sync method.

```
package samples.contact;

import java.util.Iterator;
import java.util.List;
import java.util.Map;

import flex.data.ChangeObject;
import flex.data.DataSyncException;

public class ContactAssembler{

  public List loadContacts(){
    ContactDAO dao = new ContactDAO();
    return dao.getContacts();
  }

  public List loadContacts(String name){
    ContactDAO dao = new ContactDAO();
    return dao.getContacts(name);
  }

  public Contact getContact(Map uid){
    ContactDAO dao = new ContactDAO();
    return dao.getContact(((Integer) uid.get("contactId")).intValue());
  }

  public List syncContacts(List changes){
    Iterator iterator = changes.iterator();
    ChangeObject co;
    while (iterator.hasNext()){
      co = (ChangeObject) iterator.next();
      if (co.isCreate()){
        doCreate(co);
      }
      else if (co.isUpdate()){
        doUpdate(co);
      }
      else if (co.isDelete()){
        doDelete(co);
      }
    }
    return changes;
  }

  private void doCreate(ChangeObject co){
    ContactDAO dao = new ContactDAO();
```

```
      Contact contact = dao.create((Contact) co.getNewVersion());
      co.setNewVersion(contact);
      }

    private void doUpdate(ChangeObject co){
      ContactDAO dao = new ContactDAO();
      try{
        dao.update((Contact) co.getNewVersion(), (Contact)
          co.getPreviousVersion());
      }
      catch (ItemNotFoundException e){
        throw new DataSyncException(co);
      }
    }

    private void doDelete(ChangeObject co){
      ContactDAO dao = new ContactDAO();
      try{
        dao.delete((Contact) co.getNewVersion(), (Contact)
          co.getPreviousVersion());
      }
      catch (ItemNotFoundException e){
        throw new DataSyncException(co);
      }
    }
}
```

Depending on the type of data change found, the `syncContacts()` method calls the `doCreate()`, `doUpdate()`, or the `doDelete()` method, which in turn attempts to create, update, or delete a data item. In the DAO, a client's previous version of the data to be changed is compared to what is currently in the back-end data resource. A DataSyncException is thrown if the previous version of the data does not match what is currently in the data resource. A DataSyncException results in a DataConflictEvent on the client, which you can handle in ActionScript code on the client. For more information about conflict resolution on the client, see "Handling data synchronization conflicts" on page 1514.

The following example shows the `update()` and `delete()` methods in the DAO, which compare the previous version of data from the client to what's current in the SQL database that is the data resource for this destination. If the previous version of data from the client does not match what is in the database, the data is not changed, and an exception is thrown, which results in a DataConflictEvent on the client.

```
...
public void update(Contact newVersion, Contact previousVersion) throws
  DAOException, ItemNotFoundException{
    Connection c = null;
    try{
      c = ConnectionHelper.getConnection();
```

```
      PreparedStatement ps = c.prepareStatement("UPDATE contact SET
        first_name=?, last_name=?, address=?, city=?, zip=?, state=? WHERE
        contact_id=? AND first_name=? AND last_name=? AND address=? AND
        city=? AND zip=? AND state=?");
      ps.setString(1, newVersion.getFirstName());
      ps.setString(2, newVersion.getLastName());
      ps.setString(3, newVersion.getAddress());
      ps.setString(4, newVersion.getCity());
      ps.setString(5, newVersion.getZip());
      ps.setString(6, newVersion.getState());
      ps.setInt(7, newVersion.getContactId());
      ps.setString(8, previousVersion.getFirstName());
      ps.setString(9, previousVersion.getLastName());
      ps.setString(10, previousVersion.getAddress());
      ps.setString(11, previousVersion.getCity());
      ps.setString(12, previousVersion.getZip());
      ps.setString(13, previousVersion.getState());
      if (ps.executeUpdate() == 0){
        throw new ItemNotFoundException("Item not found");
      }
    }
    catch (Exception e){
      e.printStackTrace();
      throw new DAOException(e);
    }
    finally{
      ConnectionHelper.closeConnection(c);
    }
  }

  public void delete(Contact newVersion, Contact previousVersion) throws
  DAOException, ItemNotFoundException{
    Connection c = null;
    try{
      c = ConnectionHelper.getConnection();
      PreparedStatement ps = c.prepareStatement("DELETE FROM contact WHERE
        contact_id=? AND first_name=? AND last_name=? AND address=? AND
        city=? AND zip=? AND state=?");
      ps.setInt(1, newVersion.getContactId());
      ps.setString(2, previousVersion.getFirstName());
      ps.setString(3, previousVersion.getLastName());
      ps.setString(4, previousVersion.getAddress());
      ps.setString(5, previousVersion.getCity());
      ps.setString(6, previousVersion.getZip());
      ps.setString(7, previousVersion.getState());
      if (ps.executeUpdate() == 0){
        throw new ItemNotFoundException("Item not found");
      }
    }
    catch (Exception e){
```

```
        e.printStackTrace();
        throw new DAOException(e);
}
      finally{
        ConnectionHelper.closeConnection(c);
      }
   }
...
```

NOTE

When you compile code that uses Flex Data Services Java APIs, you must include the messaging.jar and flex-messaging-common.jar files in your classpath.
The compiled assembler class and any other required classes, such as a DAO class, must be available in the web application's classpath.

Using the Assembler interface approach

To use the Assembler interface approach in an assembler, you must implement the flex.data.assemblers.Assembler interface. The flex.data.assemblers.AbstractAssembler class extends this interface, so extending this class is the easiest way to write your own assembler.

Depending on the fill request that a client sends, the `fill()` method performs a specific query to create the Collection object that is returned to the client side. An assembler that implements the Assembler interface has an `autoRefreshFill()` method, which returns a Boolean value that determines whether to refresh fills when data items change. The `autoRefreshFill()` method of the AbstractAssembler class returns `true`.

When the assembler's `autoRefreshFill()` method returns `true`, the `refreshFill()` method is called for each item that changes. The `autoRefreshFill()` method can return the following values:

Value	Usage
`DO_NOT_EXECUTE_FILL`	Use this value when you want to leave the contents of the fill unchanged.
`EXECUTE_FILL`	Use this value when you want the data service to call your fill method.
`APPEND_TO_FILL`	Use this value when you just want this changed item to be added onto the end of the list returned by the last fill invocation.

The following example shows the source code for the CompanyAssembler class, which is part of the CRM application included in the samples web application. The source code for the application's Java assemblers and DAO classes are in the WEB_INF/src/samples/crm directory of the samples web application. The CompanyAssembler class uses the Assembler interface approach. It extends the flex.data.assemblers.AbstractAssembler class and does not override the `autoRefreshFill()` method, which returns `true`. It implements a `refreshFill()` method; depending on the type of data change, it executes a fill, appends to a fill, or does not execute a fill.

```
package samples.crm;

import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Collection;
import java.util.Arrays;

import flex.data.ChangeObject;
import flex.data.DataSyncException;
import flex.data.assemblers.AbstractAssembler;

public class CompanyAssembler extends AbstractAssembler
{
  public Object getItem(Map uid)
  {
    CompanyDAO dao = new CompanyDAO();
    return dao.getCompany(((Integer) uid.get("companyId")).intValue());
  }

    /**
     * This method is called by the Data Management Service for any fill
     * methods called by the client that are not configured in the
     * configuration file explicitly using the fill-method tag. Those fill
     * methods will call those methods directly.
     * This can be used either as a replacement
     * to that mechanism or to extend it to have additional fill methods.
     * </p>
     * <p>
     * Depending upon the number and type of fills you expect, it is
     * a good practice to use the first parameter as the name of the
     * query and the rest as the values when you have lots of queries.
     */
    public Collection fill(List fillParameters)
    {
    CompanyDAO dao = new CompanyDAO();
        if (fillParameters.size() == 0)
            return dao.findCcompanies(null, null);
        else if (fillParameters.size() == 1)
```

```
            return dao.findCompanies((String) fillParameters.get(0), null);
        else if (fillParameters.size() == 2)
            return dao.findCompanies((String) fillParameters.get(0),
                                     (String) fillParameters.get(1));

        return super.fill(fillParameters);
    }
  /**
    * If your fill methods are auto-refreshed, this method is called
    * for each item that changes (either created or updated as indicated
    * by the isCreate parameter). It can return DO_NOT_EXECUTE_FILL if
    * you want to leave the contents of the fill unchanged, EXECUTE_FILL if
    * you want the data service to call your fill method, or APPEND_TO_FILL
    * if you just want this changed item to be added onto the end of the
    * list returned by the last fill invocation.
    *
    * @param fillParameters the client side parameters to a fill method
    * that created a managed collection still managed by one or more
    * clients.
    * @param the item which is being created or updated
    * @param true if this is a create operation, false if it is an update.
    * @return DO_NOT_EXECUTE_FILL - do nothing, EXECUTE_FILL - re-run the
    * fill method to get the new list, APPEND_TO_FILL - just add it to the
    * existing list.
    */
    public int refreshFill(List fillParams, Object item, boolean isCreate)
    {
        Company company = (Company) item;
        if (fillParams.size() == 0) // This is the "all query"
            return APPEND_TO_FILL;

        if (fillParams.size() == 1)
        {
            String name = (String) fillParams.get(0);
            if (company.getName().indexOf(name) != -1)
                return APPEND_TO_FILL;
            return DO_NOT_EXECUTE_FILL;
        }
        else if (fillParams.size() == 2)
        {
            String name = (String) fillParams.get(0);
            String mseg = (String) fillParams.get(1);
            if ((name == null || company.getName().indexOf(name) != -1) &&
              (mseg == null || company.getMarketSegments().contains(mseg)))
                return APPEND_TO_FILL;
            return DO_NOT_EXECUTE_FILL;
        }
        return EXECUTE_FILL;
    }
```

```
  public void createItem(Object newVersion)
  {
    CompanyDAO dao = new CompanyDAO();
    dao.create((Company) newVersion);
  }

  public void updateItem(Object newVersion, Object prevVersion, List
  changes)
  {
    CompanyDAO dao = new CompanyDAO();
    try
    {
      dao.update((Company) newVersion, (Company) prevVersion, changes);
    }
    catch (ConcurrencyException e)
    {
           int companyId = ((Company) newVersion).getCompanyId();
      System.err.println("*** Throwing DataSyncException when trying to
  update company id=" + companyId);
           // The conflict error which contains the current
          //version of the company so the client can reconcile.
      throw new DataSyncException(dao.getCompany(companyId), changes);
    }
  }

  public void deleteItem(Object prevVersion)
  {
    CompanyDAO dao = new CompanyDAO();
    try
    {
      dao.delete((Company) prevVersion);
    }
    catch (ConcurrencyException e)
    {
           int companyId = ((Company) prevVersion).getCompanyId();
      System.err.println("*** Throwing DataSyncException when trying to
  delete company id=" + companyId);
            // The conflict error which contains the current
            version of the company so the client can reconcile.
      throw new DataSyncException(dao.getCompany(companyId), null);
    }
  }
}
```

The following example shows the `fill()`, `createItem()`, and `updateItem()` methods of the CRM application's EmployeeAssembler. This assembler also uses the Assembler interface approach, but it overrides the `autoRefreshFill()` method and returns `false`. Based on the data items that are added or updated, the `createItem()` and `updateItem()` methods refresh fills with the results of specific queries that call the `refreshFill()` method of an instance of the flex.data.DataServiceTransaction class; for more information about the flex.data.DataServiceTransaction class, see "Pushing data changes from the server to clients" on page 1569.

```
public Collection fill(List fillParameters)
{
  EmployeeDAO dao = new EmployeeDAO();
  if (fillParameters.size() == 0)
    return dao.getEmployees();

  String queryName = (String) fillParameters.get(0);

  if (queryName.equals("match"))
    return dao.findMatchingEmployees((String) fillParameters.get(1));
  if (queryName.equals("byCompany"))
    return dao.findEmployeesByCompany((Integer) fillParameters.get(1));

    return super.fill(fillParameters); // throws an error.
}

public void createItem(Object newItem)
{
  Employee newEmployee = (Employee) newItem;
  DataServiceTransaction dtx =
    DataServiceTransaction.getCurrentDataServiceTransaction();
  EmployeeDAO dao = new EmployeeDAO();
  dao.create(newEmployee);

  // Refresh the "all employees" query
  dtx.refreshFill("crm.employee", new ArrayList(0));
  // Refresh anyone who is looking at the employees for the company
  // of this employee.
  dtx.refreshFill("crm.employee", Arrays.asList(
    new Object[] {"byCompany",
      newEmployee.getCompany() == null ? null : new
      Integer(newEmployee.getCompany().getCompanyId())}));
}

public void updateItem(Object newVersion, Object prevVersion, List changes)
{
  EmployeeDAO dao = new EmployeeDAO();
  DataServiceTransaction dtx =
    DataServiceTransaction.getCurrentDataServiceTransaction();
  try
```

```
  {
    Employee newEmployee = (Employee) newVersion;
    Employee prevEmployee = (Employee) prevVersion;
    dao.update(newEmployee, prevEmployee, changes);

/*
* Now that the update has been performed, we are going to determine which
* fills need to be refreshed based on the changes that have been made.
*/
/*
* If someone has changed the company, we need to update two fills - the
* one for the old and new versions of the company
*/
    if (changes == null || changes.contains("company"))
    {
      Company prevCompany = prevEmployee.getCompany();
      Company newCompany = newEmployee.getCompany();
      dtx.refreshFill("crm.employee",
        Arrays.asList(new Object[] {"byCompany",
        prevCompany == null ? null :
        new Integer(prevCompany.getCompanyId())}));
        dtx.refreshFill("crm.employee",
        Arrays.asList(new Object[] {"byCompany",
        newCompany == null ? null : new
        Integer(newCompany.getCompanyId())}));
      }

/*
* Only because it is so easy to do, if someone happens to viewing
* a pattern match result
* by name, update their search results in real time.
*/
      if (changes == null || changes.contains("firstName") ||
        changes.contains("lastName"))
      {
      // If we changed the first name, refresh all of the search
      //queries make for employees.
        dtx.refreshFill("crm.employee", Arrays.asList
          (new Object[] {"match", null}));
      }
    }
  catch (ConcurrencyException e)
  {
    int employeeId = ((Employee) newVersion).getEmployeeId();
    System.err.println("*** Throwing DataSyncException when trying
    to update employee id=" + employeeId );
    throw new DataSyncException(dao.getEmployee(employeeId), changes);
  }
}
```

NOTE

When you compile code that uses Flex Data Services Java APIs, you must include the messaging.jar and flex-messaging-common.jar files in your classpath.
The compiled assembler class and any other required classes, such as a DAO class, must be available in the web application's classpath.

Configuration file settings

The following table describes each of the destination elements that are specific to the Java adapter. Unless otherwise indicated, these elements are subelements of the `properties/server` element. The Hibernate assembler has specialized elements that differ from those in this table; for more information, see "Hibernate configuration files" on page 1550.

Element	Description

`<source>`
`samples.contact.ContactAssembler`
`</source>`
`<scope>application</scope>`
	The `source` element specifies the assembler class, which is the class that handles data changes. The `scope` element specifies the scope of the assembler; the valid values are `application`, `session`, and `request`.
	These elements are subelements of the `properties` element.

Element	Description
<fill-method> <name>loadOrderedPatternGroups</name> <params> java.lang.Boolean,java.lang.String </params> <ordered>true</ordered> <fill-contains-method> <name>orderedPatternContains</name> </fill-contains-method> <security-constraint ref="sample-users"/> </fill-method>	The fill-method element specifies a method to call when a fill request is made. The required name element designates the name of the method to call. The optional params element designates input parameter types for the fill operation, and accommodates method overloading. The optional fill-contains-method element points to a method in the Assembler class that takes a changed or created item and the List of fill parameters and returns true or false depending on whether the item is in that fill. If the fill is not ordered and the method returns true, the item is simply added to the end of the list of managed items for that fill. If the fill is ordered and you return true, the fill method is re-executed to get the proper ordering of the list. If the method returns false, the fill-method is left as is for that update. The optional security-constraint and security-run-as elements reference a security setting in the services-config.xml file. As an alternative to declaring fill and sync methods in fill-method and sync-method elements in the destination, you can implement the flex.data.assemblers.Assembler interface and include fill() method implementations in your assembler. For more information, see "Using the Assembler interface approach" on page 1538.

Element	Description

`<sync-method>`
`  <name>syncCustomer</name>`
`  <security-constraint ref="admins"/>`
`</sync-method>`
	The `sync-method` element specifies a method to be invoked for update, delete, and insert operations.
	The required `name` element specifies the name of the method to be invoked.
	There is no `params` element because the parameter is predefined as a List of ChangeObjects.
	The `security-constraint` and `security-run-as` elements reference a security setting in the services-config.xml file.
	As an alternative to declaring fill and sync methods in `fill-method` and `sync-method` elements in the destination, you can implement the flex.data.assemblers.Assembler interface and include `fill()` method implementations in your assembler. For more information, see "Using the Assembler interface approach" on page 1538.

`<get-method>`
`  <name>getCustomer</name>`
`  <security-constraint ref="admins"/>`
`</get-method>`
	The `get-method` element specifies a method to call to retrieve a single item instance instead of a List of item instances. If present, this element always takes a single parameter, which is the Map of values used to denote object identity.

`<count-method>`
`  <name>numCustomers</name>`
`  <security-constraint ref="admins"/>`
`</count-method>`
	The `count-method` element specifies a method to call to retrieve an integer from the assembler that can be used to return a count of items without first loading all of a data set into memory. Like the `fill-method` element, the `count-method` element can accept parameters. This method does not retrieve the size of a filled sequence, in that any given sequence can represent a subset of data, and sequences are always fully loaded into the server. Count method implementations may execute a COUNT statement against a database.

`<properties>`
`  <item-class>`
`  mypackage.MyClass`
`  </item-class>`
`...`
`</properties>`
	Converts anonymous ActionScript objects received from the client side back into a single strongly typed Java class that the Java assembler expects.
	Note that the `item-class` element is a subelement of the `properties` element.

Using a Hibernate destination

The Hibernate assembler is a specialized Java assembler class that you use with the Java adapter to provide a bridge to the Hibernate object–relational mapping system. The Hibernate assembler supports Hibernate 3.0 and later. Using this assembler, you do not have to write Data-Management-specific Java code to integrate with an existing Hibernate back end. Instead, you configure Hibernate XML-based mapping and configuration files, and then configure a Data Management Service destination that references those files. The Hibernate assembler uses the Hibernate configuration files at run time by using Hibernate APIs to persist data changes into a relational database. Hibernate operations are encapsulated within the assembler. Associations in your Hibernate configuration should be mirrored as associations in your Flex destination definition.

The Data Management Service count, get, create, update, and delete methods all correspond to Hibernate operations of similar names and behavior, and they require no configuration other than the configuration in the Hibernate mapping file. However, the object retrieval methods of Hibernate are sophisticated, and you must include additional configuration to map a Data Management Service fill method to one of the Hibernate object retrieval methods.

The following example shows a destination that uses the Hibernate assembler. This HibernatePerson destination has a many-to-one relationship with a HibernateGroup destination.

```
...
<destination id="HibernatePerson" channels="rtmp-ac">
  <adapter ref="java-adapter" />
  <properties>
    <source>flex.data.assemblers.HibernateAssembler</source>
    <scope>application</scope>
    <metadata>
      <identity property="id"/>
      <many-to-one property="group"
        destination="HibernateGroup" lazy="true"/>
      </metadata><network>
      <paging enabled="true" pageSize="10" />
      <throttle-inbound policy="ERROR" max-frequency="500"/>
      <throttle-outbound policy="REPLACE" max-frequency="500"/>
    </network>
    <server>
      <!-- If this element is not present, the adapter assumes that
      the Hibernate entity has the same name as the destination id.
      -->
      <hibernate-entity>contacts.hibernate.Person</hibernate-entity>
      <!-- Conflict modes determine whether to check for conflicts at all
      and if so whether to check on changed properties only or whether
```

```
        to verify that the client had correct data for the entire object.
        Valid values for update-conflict-mode are NONE, PROPERTY, and OBJECT.
        Valid values for delete-conflict-mode are NONE and OBJECT.
        -->
        <update-conflict-mode>PROPERTY</update-conflict-mode>
        <delete-conflict-mode>OBJECT</delete-conflict-mode>
        <fill-configuration>
          <use-query-cache>false</use-query-cache>
          <allow-hql-queries>true</allow-hql-queries>
        </fill-configuration>
      </server>
    </properties>
</destination>
```

The following examples shows a corresponding ActionScript method for working with the HibernatePerson and HibernateGroup destinations:

```
...
private function createDataCollection():void {
  groupDS = new DataService("HibernateGroup");
  groupDS.addEventListener(MessageFaultEvent.FAULT, faultHandler);
  groupDS.addEventListener(MessageEvent.RESULT, resultHandler);
  groupDS.addEventListener(DataConflictEvent.CONFLICT, conflictHandler);
  groupDS.autoCommit = false;
  personDS = new DataService("HibernatePerson");
  personDS.autoCommit = false;
  personDS.fill(allPersons,"flex:hql","From Person where groupId is null");
  groupDS.fill(groups1,"flex:hql","From Group");
  personDS.dataStore = groupDS.dataStore;
}
...
```

Supported Hibernate features

The Hibernate assembler currently supports the following Hibernate features for working with data:

Hibernate feature	Description
Load and get	Retrieves a single object by identity. The Data Management Service `getItem()` functionality corresponds to the Hibernate get functionality. The Hibernate load method is not employed by the Hibernate assembler.

Hibernate feature	Description
Hibernate Query Language (HQL)	Retrieves a List of items based on an HQL query. You can configure the Data Management Service fill functionality to recognize `fill` arguments as HQL query strings along with a map of parameters. To use HQL queries from your client code, you must first set `<allow-hql-queries>true</allow-hql-queries>` in the `fill-configuration` section. **Note:** By allowing clients to specify arbitrary HQL queries, you are exposing a potentially dangerous operation to untrusted client code. Clients may be able to retrieve data you did not intend for them to retrieve or execute queries that cause performance problems in your database. Adobe recommends that you use the HQL queries for prototyping, but replace them with named queries before your deploy your application and use the default setting `<allow-hql-queries>false</allow-hql-queries>` in your production environment. When using HQL, the client-side `DataService.fill()` method would look similar to the following example: `myService.fill(myCollection, "flex:hql", ["from Person p where p.firstName = :firstName", parameterMap]);` The `parameterMap` parameter contains the parameter name and value bindings for the query. The preceding example uses named parameters. Positional parameters are also supported; that is, you could write the HQL string as `"from Person p where p.firstName = ?"`. In that case, the `parameterMap` parameter is an array of values instead of a map of values. The use of positional parameters instead of named parameters is generally considered a less reliable approach, but both approaches are supported.
Named Hibernate queries	In Hibernate, you configure a named query by using the Hibernate configuration element named `query`. You can then call this query from a Flex client by passing the name of the Hibernate query as the second argument to a `DataService.fill()` method. As with HQL queries, if the Hibernate query is using named parameters, you specify this on the client with an anonymous object that contains those parameters. If your Hibernate query is using positional parameters, you provide an array of parameters. The following example shows a `fill()` method that uses a named query: `myDS.fill(myCollection, "eg.MyHibernateEntity.myQueryName", myQueryParameterObjectOrArray);`

Hibernate configuration files

Hibernate uses two types of configuration files. The global configuration file contains declarations for database settings and behaviors. The mapping file contains declarations for how Java classes and fields map to database tables and columns. The Data Management Service must have access to these files at run time. Hibernate requires that the files are accessible in the web application classpath. You should place the hibernate.cfg.xml file in the WEB_INF/classes directory and the Hibernate JAR file in the WEB_INF/lib directory of your web application.

The Hibernate assembler uses the Hibernate mappings file to expose named queries to client-side code. Your client code can provide parameters to these queries by using fill parameter arguments. You can also optionally allow clients to use HQL queries by setting the `allow-hql-queries` element to `true` in the destination configuration.

Flex supports Hibernate `version` or `timestamp` elements for automatic concurrent conflict detection in Hibernate entity mappings. If these settings are used in a Hibernate configuration file, they take precedence over any FDS undate/delete conflict modes defined. A warning is logged that the automatic Hibernate conflict detection will be used if there are also FDS conflict modes defined.

NOTE	If you use this option, you are exposing arbitrary queries to be executed by untrusted code. Use this option only for prototyping and switch to named queries before deploying your production application.

Currently the Hibernate assembler does not support criteria-based queries but you can extend the flex.data.assemblers.HibernateAssembler class and expose whatever queries you like by writing some additional Java code. This class is documented in the public Flex Data Services Javadoc documentation.

It is possible to use the Hibernate assembler to persist data to a relational database with no configuration in the server section of a destination. In this scenario, fill requests are mapped to HQL queries, and the Hibernate assembler does not use the query cache or Hibernate paging. However, the Hibernate configuration files must be available to the Flex Data Services web application at run time.

To expose custom handling of type mappings on query parameter bindings, Flex honors the Hibernate type mappings provided in the Hibernate configuration file for a persistent Java object. The values for the `type` attribute are treated as Hibernate types that map Java types to SQL types for all supported databases. If omitted, Hibernate makes a choice based on the Java type of the parameter. In some cases, as in dealing with mapping Java Date types to database date or timestamp types, this approach should most likely be overridden with a parameter declaration in the Hibernate mapping.

The following example shows a Hibernate mapping file, Group.hbm.xml, which maps to a Java class named Group. This file declares a one-to-many relationship between a Group object and a Person object:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="dev.contacts.hibernate.Group" table="simplegroup">
        <id name="id" column="id">
            <generator class="identity"/>
        </id>
        <property name="name" />
        <many-to-one name="leader"
         class="dev.contacts.hibernate.Person" column="leaderId"
         cascade="save-update"/>
        <set name="members" inverse="true">
        <key column="groupId" />
        <one-to-many class="dev.contacts.hibernate.Person" />
        </set>
    </class>
</hibernate-mapping>
```

The following example shows the source code for the corresponding Person class:

```
package contacts.hibernate;
import java.util.List;
import java.util.HashSet;
import java.util.Set;

public class Group {
    private String name = null;
    private Set members = null;
    private Person leader = null;
    private String id;
    public String getName() {
        return name;
    }
    public void setName(String n) {
        name = n;
    }
    public Set getMembers() {
        return members;
    }
    public void setMembers(Set s) {
        members = s;
    }
    public Person getLeader() {
        return leader;
```

```
    }
    public void setLeader(Person sp) {
        leader = sp;
    }

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }

    public int hashCode() {
        if (id == null)
            return 0;
        return id.hashCode();
    }

    public boolean equals(Object o) {
        if (o instanceof Group) {
            Group e = (Group) o;
            if (e.getId() == getId())
                return true;
            if (e.getId() == null)
                return false;
            return e.getId().equals(getId());
        }
        return false;
    }
}
```

Configuration file settings

The following table describes each of the destination elements that are specific to the Hibernate assembler. These elements are children of the `server` element.

Element	Description

`<source>`
`flex.data.assemblers.HibernateAssembler`
`</source>`
`<scope>application</scope>`
	The `source` element specifies the assembler class, which is the class that handles data changes. The `scope` element specifies the scope of the assembler; the valid values are `application`, `session`, and `request`. These elements are subelements of the `properties` element.

`<hibernate-entity>`
`Person`
`</hibernate-entity>`
	Exposes custom handling of type mappings on query parameter bindings, Flex honors the Hibernate type mappings provided in the Hibernate configuration for a persistent Java object.
	To gain access to the class metadata for such programmatic query creation and parameter type mapping, you can configure a destination with the name of the Hibernate entity that represents the persistent Java object.
	If omitted from the configuration, the Hibernate entity type is assumed to be the same as the destination `id`.

`<update-conflict-mode>`
`PROPERTY`
`</update-conflict-mode>`
	Verifies that update data from a Flex client application is not stale. If omitted, no data verification occurs. Valid values are `NONE`, `PROPERTY`, and `OBJECT`. A data conflict results in a DataConflictEvent on the client.

`<delete-conflict-mode>`
`OBJECT`
`</delete-conflict-mode>`
	Verifies that delete data from a Flex client application is not stale. If omitted, no data verification occurs. Valid values are `NONE` and `OBJECT`. A data conflict results in a DataConflictEvent on the client.

Element	Description

`<fill-configuration>`
`  <allow-hql-queries>true</allow-hql-queries>`
`  <use-query-cache>true</use-query-cache>`
`</fill-configuration>`
	The `allow-hql-queries` element determines whether HQL queries can be used. The default value is `false`. The `use-query-cache` element is an option to the Hibernate query method which allows Hibernate to cache queries. The default value is `false`.

`<create-security-constraint ref="sample-users"/>`	Applies the security constraint referenced in the `ref` attribute to create requests.
`<update-security-constraint ref="sample-users"/>`	Applies the security constraint referenced in the `ref` attribute to update requests.
`<delete-security-constraint ref="sample-users"/>`	Applies the security constraint referenced in the `ref` attribute to delete requests.

Using the ActionScript object adapter

The ActionScript object adapter is intended for scenarios that require only transient data and no back-end data resource, and a Flex client application is the only creator and consumer of data objects. The Data Management Service uses the ActionScript object adapter to manage objects in memory. You can use any field in the ActionScript object to provide unique identity in the `identity` element of the destination's metadata section. You can establish composite identity by specifying more than one field. If unspecified, by default the `uid` property of the ActionScript object provides the object identity.

The following table describes each of the destination elements that are specific to the ActionScript object adapter. These elements are children of the `server` element.

Element	Description
`<create-security-constraint` `  ref="sample-users"/>`	Applies the security constraint referenced in the `ref` attribute to the create requests.
`<read-security-constraint` `  ref="sample-users"/>`	Applies the security constraint referenced in the `ref` attribute to fill requests.
`<update-security-constraint` `  ref="sample-users"/>`	Applies the security constraint referenced in the `ref` attribute to update requests.
`<delete-security-constraint` `  ref="sample-users"/>`	Applies the security constraint referenced in the `ref` attribute to delete requests.

The following example shows a client application that uses the ActionScript adapter to persist a single data item if it doesn't already exist in a TextArea control across all clients. When the server is stopped, the data is lost because there is no back-end data resource.

```
<?xml version="1.0"?>
<!-- fds\datamanagement\ASAdapter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
 height="100%" width="100%"
 creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
            import mx.data.DataService;
            import mx.rpc.AsyncToken;

            public var noteObj:Object = new Object();
            [Bindable]
            public var getToken:AsyncToken;
            private var ds:DataService;

            private function initApp():void {
            ds = new DataService("notes");
            ds.autoCommit = false;
            noteObj.noteId = 1;
            noteObj.noteText =
                "Type your notes here and share them with other clients!";
            getToken = ds.getItem(noteObj, noteObj);
            }
      ]]>
    </mx:Script>
    <mx:Binding source="log.text" destination="getToken.result.noteText"/>
    <mx:TextArea id="log" width="100%" height="100%"
text="{getToken.result.noteText}"/>
    <mx:Button label="Send" click="ds.commit();"/>
</mx:Application>
```

The only server-side configuration is the following destination in the data-management_config.xml file:

```
<destination id="notes">
  <adapter ref="actionscript"/>
    <properties>
     <metadata>
       <identity property="noteId"/>
     </metadata>
    </properties>
</destination>
```

Each item must have a unique ID and there is no back-end data resource that automatically creates IDs. Therefore, for an application in which you require an ArrayCollection data items, you must generate item IDs on the client when you create managed objects. The code in the following example uses the `Math.random()` method to generate a `customer.custId`. In a real application, `custId` would be the property specified as the identifier in a destination in the configuration file.

```
private function newCustomer():void {
  dg.selectedIndex = -1;
  customer = new Customer();
  customer.custId = Math.random() * 1000;
}
```

Managing hierarchical collections

A *hierarchical collection* is one that contains an object that has complex properties of types other than simple primitive values. For example, a collection might contain a Person object that has an Address property, which has street, city, and state properties.

There are three techniques for managing hierarchical collections in conjunction with the Data Management Service:

- The *hierarchical values* approach in which a single Data Management Service destination manages an entire tree of data.
- The *managed associations* approach in which you define a parent destination that has declared associations to child destinations, which manage the values of the properties of the child objects.
- The *query* approach in which you can implement a relationship between two objects using queries where one of the parameters of a fill method defined in an assembler is the ID of the related object.

You can use these approaches independently, or in combination on a property-by-property basis.

In the hierarchical values approach and the managed associations approach, the parent destination returns the same graph of objects from its fill invocation. In the hierarchical values approach, the parent object's adapter is responsible for saving the state of the entire complex property. This is the default behavior. The parent is also responsible for updating the entire graph in response to a sync invocation; on any change, it gets the complete new graph for the object and is responsible for updating any change in that graph.

In the managed associations approach, the destination configuration for the parent destination contains an additional tag that defines an association for the child object that has its own destination. This association tag refers to the separate destination that is responsible for managing the state of the child object.

When you use the hierarchical data approach or the managed association approach, you must mark each class in the top-level object's graph of managed classes with the `[Managed]` metadata tag or explicitly implement the mx.data.IManaged interface. This ensures that the managed objects support the proper change events and that getter methods for one-to-one association properties that are lazily loaded can throw ItemPendingErrors. For more information about lazy loading and ItemPendingErrors, see "Implementing a managed association" on page 1559.

The following example shows the code for an ActionScript class that uses the `[Managed]` metadata tag. This Employee class is used in an ArrayCollection of Employees, which is used as a property of a Company class.

```
package samples.crm
{

  import mx.data.IManaged;
  import mx.data.utils.Managed;
  import mx.core.mx_internal;

  [Managed]
  [RemoteClass(alias="samples.crm.Employee")]
  public class Employee{
    public var employeeId:int;

    public var firstName:String = "";

    public var lastName:String = "";

    public var title:String = "";

    public var phone:String = "";

    public var email:String = "";
  }
}
```

The following example shows a class that implements the mx.data.IManaged interface instead of using the `[Managed]` metadata tag. When you implement mx.data.IManaged, you also must support the mx.core.IUID interface, which requires a `uid` property. You also must include the `referenceIds` and `destination` variables declared under the last comment block in this class.

```
import mx.core.mx_internal;
import mx.data.Managed;
import mx.data.IManaged;
import mx.utils.UIDUtil;

  [RemoteClass(alias="foo.bar.Customer")]
  public class Customer implements IManaged {
    public function Customer() {
    super();
  }

  [Bindable(event="propertyChange")]
  public function get firstName():String {
    _firstName = Managed.getProperty(this, "firstName", _firstName);
    return _firstName;
  }

  public function set firstName(value:String):void {
    var oldValue:String = this._firstName;
    _firstName = value;
    Managed.setProperty(this, "firstName", oldValue, _firstName);
  }
// all implementors of IManaged must support IUID which requires a uid
// property
  [Bindable(event="propertyChange")]
  public function get uid():String {
    // If the uid hasn't been assigned a value, just create a new one.
    if (_uid == null) {
      _uid = UIDUtil.createUID();
    }
    return _uid;
  }

  public function set uid(value:String):void {
    _uid = value;
  }
// These are special requirements of any object that wants to be
// "managed" by a DataService.
// The referencedIds is a list of object IDs that belong to properties that
// are lazily loaded.
// The destination is used to look up metadata information about the
// associations this object has been configured with in the
// flex-dataservices.xml file located at the remote destination.
```

```
    mx_internal var referencedIds:Object = {};
    mx_internal var destination:String;

    private var _firstName:String;
    private var _uid:String;
}
```

Implementing a managed association

In the managed association approach, the parent destination is not responsible for processing updates for changes made to properties of the child objects. It is only responsible for processing the changes to the `id` values of children referenced by the parent. However, if a Person object has an Address property, using the hierarchical values approach, a change to the city of a person's address would be handled by the Person destination.

In the managed association approach, the Address destination would process a change to the city. The Person destination would process a change if someone assigned an Address with a different `id` to the Address property of the Person object. You can also configure whether the relationship supports lazy loading of object graphs from the server to the client.

You can configure relationships between destinations that are unidirectional one-to-one, one-to-many, or many-to-one relationships, or bidirectional many-to-many relationships. You configure managed associations in `one-to-one`, `one-to-many`, and `many-to-many` elements in a destination. These elements are subelements of the `metadata` element, which is a subelement of the `properties` element.

When working with many-to-many relationships for managed associations that rely on database tables, you must create a join table rather than using a foreign key in one of the tables to establish the relationship between the tables that the destinations use.

You set the `lazy` attribute of a `one-to-one`, `one-to-many`, or `many-to-many` element to `true` to enable the lazy loading of object graphs from the server to clients. When the client accesses the value for the first time, an ItemPendingError is thrown either from the `ArrayCollection.getItem()` method (for multiple valued associations) or the `Managed.getProperty()` method for single-valued associations. This causes a request to fetch the referenced object, and an event handler in the ItemPendingError is invoked to notify the client when the item is available. When you use Flex data binding, this behavior happens automatically when you set the `lazy` attribute to `true`; the data appears when it is needed.

If you do not use data binding in conjunction with lazy loading, you can catch the ItemPendingError in your code and handle it accordingly. An ItemPendingError is thrown when retrieving an item from ArrayCollection requires an asynchronous call. If the receiver of this error needs notification when the requested item when the asynchronous call completes, it must use the `addResponder()` method and specify an object that supports the mx.rpc.IResponder interface to respond when the item is available. The mx.collections.ItemResponder class implements the IResponder interface and supports a `data` property.

In the following example, an ItemPendingError is thrown and caught when a group object is not yet available to a groupCollection object on the client. The groupCollection object is filled from the `AssocGroup` destination on the server. The destination has an `admin` property that is a lazily loaded reference to an AssocPerson instance. The `printAdminName()` method is called for the first group in the groupCollection object. If a group is not available, it throws an ItemPendingError. An ItemResponder is registered to handle the ItemPendingError; the ItemResponder calls the `printAdminName()` method when the group is available. If there is a fault, the ItemResponder calls the `fetchAdminError()` method.

```
...
import mx.collections.ItemResponder;
import mx.messaging.messages.ErrorMessage;
import mx.collections.errors.ItemPendingError;
...

public function printAdminName (data:Object, group:Object):void {
  trace(group.admin.firstName);
}

public function fetchAdminError(message:ErrorMessage):void {
  trace("error occurred fetching admin: " + message.faultString);
}

// This function is called to retrieve a "group" object
// that has a lazily loaded reference to a Person in its admin property.
// The printAdminName function may throw the ItemPendingError
// when it retrieves the group.admin property if that object has
// not yet been loaded by the client.
// The function registers a listener that calls printAdminName
// again when the value is available.

public function dumpAdminName():void {
  try {
    printAdminName(null, groupCollection.getItemAt(0));
  }
  catch (ipe:ItemPendingError) {
    trace("item pending error fetching admin.");
    ipe.addResponder(new ItemResponder(printAdminName, fetchAdminError,
```

```
    groupCollection.getItemAt(0)));
  }
}
...
```

In summary, for the hierarchical values approach, a single destination manages the entire state of the graph. However, for the managed associations approach, the parent destination manages the IDs of the referenced child objects and the relationship between the parent and child objects. For the managed associations approach, the child assembler owns the state of the child objects themselves. A side benefit of using the managed association approach is that you can use destinations independently of each other in one application, while using them together in another application.

Looking at a managed association example

This section describes some of the key aspects of an application that uses the managed association approach to model a one-to-many relationship between companies and employees.

On the client side of the application, a DataService component named dsCompany manages an ArrayCollection of Company objects. The dsCompany component uses a server-side destination named company.

Each Company object contains an `employees` property, which is an ArrayCollection of Employee objects. These client-side Company and Employee objects are mapped to server-side Java classes of the same name, which are populated by method calls to the company destination.

Another destination, named employee, provides the employee data for each company. The company destination contains a one-to-many relationship between the company and employee destinations. Each destination has its own assembler and DAO class.

Because of the one-to-many relationship, the employee destination manages all changes to employee data with its own assembler and DAO independently of the company destination. Company and Employee Java classes represent companies and employees, and these classes are mapped to corresponding Company and Employee ActionScript classes on the client side.

Destination configuration

The following example shows relevant sections of the company and employee destinations, including the `one-to-many` element that defines the relationship between the destinations. The `one-to-many` element defines a one-to-many-relationship between the company destination's `employees` property and the employee destination. The boldface text highlights the configuration for the one-to-many relationship.

```
...
<destination id="company">
  <adapter ref="java-adapter"/>
  <properties>
    <source>samples.crm.CompanyAssembler</source>
    <scope>application</scope>

    <metadata>
      <identity property="companyId"/>
      <one-to-many property="employees" destination="employee"/>
    </metadata>
...
      <server>
        <fill-method>
          <name>loadCompanies</name>
        </fill-method>
        <sync-method>
          <name>syncCompanies</name>
        </sync-method>
...
  </properties>
</destination>
...
<destination id="employee">
  <adapter ref="java-adapter"/>
  <properties>
      <source>samples.crm.EmployeeAssembler</source>
      <scope>application</scope>

    <metadata>
      <identity property="employeeId"/>
    </metadata>

    <server>
      <fill-method>
        <name>loadEmployees</name>
      </fill-method>

      <sync-method>
        <name>syncEmployees</name>
      </sync-method>
...
```

```
    </server>
  </properties>
</destination>
```

ActionScript code

The following example shows some of the ActionScript code for the client-side application that fills an ArrayCollection with Company objects. An ActionScript class named Company contains an `employees` property that is an ArrayCollection of Employee objects (see `var company` in the example).

Company and Employee ActionScript objects are mapped to corresponding Company and Employee Java objects on the server side. The Employee ActionScript class includes the `[Managed]` metadata tag above its class declaration to indicate that it is a managed class. The `conflictHandler()` method handles data synchronization conflicts from both the company and employee destinations because of the one-to-many relationship between the two destinations by listening for conflict events on the data store. Because there is an association between the company and employee destinations, by default they share the same `dataStore` property. Conflicts and uncommitted changes are kept in the data store to maintain data integrity when you have associations.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
  creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
...
    [Bindable]
    public var companies:ArrayCollection;
    [Bindable]
    public var company:Company;
    private var dsCompany:DataService;

    private function initApp() {
      companies = new ArrayCollection();
      dsCompany = new DataService("company");
...
      dsCompany.dataStore.addEventListener(DataConflictEvent.CONFLICT,
        conflictHandler);
      dsCompany.autoCommit = false;
...
    }
...
    private function conflictHandler(event:DataConflictEvent):void {
      Hourglass.remove();
      var conflicts:Conflicts = dsCompany.dataStore.conflicts;
      var c:Conflict;
      for (var i:int=0; i<conflicts.length; i++) {
```

```
            c = Conflict(conflicts.getItemAt(i));
            Alert.show("Reverting to server value", "Conflict");
            c.acceptServer();
          }
        }
    </mx:Script>
</mx:Application>
```

MXML code

When the application first loads, a call to `dsCompany.fill()` method loads all of the company objects. Because the `one-to-many` association element for the `employees` property does not have a `lazy` attribute set to `true`, each company also loads all of the employees for the company. In other words, for the entire list of companies, and for each company, the entire list of employees are fetched in one request as part of the fill call.

If you set the value of the `lazy` attribute to `true` for the employees association, only the IDs of the employees would be loaded with the companies. The data for an individual employee would be fetched when that element of the company.employees property is accessed. For efficiency, when one employee is requested, an entire page of employees is fetched at the same time. If you set `pageSize` to `0` (the default), when the first employee is requested, all employees in that ArrayCollection are fetched. If you set `pageSize` to `1`, each employee would be loaded individually when it is requested.

The following example shows the MXML code for the DataGrid control that displays the company summary data:

```
...
<mx:DataGrid id="dg" dataProvider="{companies}"
  width="100%" height="100%" change="companyChange()">
  <mx:columns>
    <mx:DataGridColumn dataField="companyId" headerText="Id"/>
    <mx:DataGridColumn dataField="name" headerText="Company Name"/>
    <mx:DataGridColumn dataField="city" headerText="City"/>
  </mx:columns>
</mx:DataGrid>
...
```

The following example shows the MXML code that displays the company details. An editable DataGrid of the company employees appears in its own tab; because of the one-to-many relationship between the company and employee destinations, changes to the employee data are sent to the employee destination.

```
...
<mx:TabNavigator width="100%" height="100%">
  <mx:Form width="100%" height="100%" label="General">
    <mx:FormItem label="Company Name" required="true">
      <mx:TextInput id="companyName" width="250" text="{company.name}"/>
```

```
    </mx:FormItem>
     <mx:FormItem label="Address">
      <mx:TextInput id="address" width="250" text="{company.address}"/>
     </mx:FormItem>
     <mx:FormItem label="City">
       <mx:TextInput id="city" width="250" text="{company.city}"/>
     </mx:FormItem>
     <mx:FormItem label="State">
     <mx:TextInput id="state" width="250" text="{company.state}"/>
     </mx:FormItem>
     <mx:FormItem label="Zip">
       <mx:TextInput id="zip" width="250" text="{company.zip}"/>
     </mx:FormItem>
   </mx:Form>

   <mx:VBox label="Employees" width="100%" height="100%">
     <mx:HBox>
       <mx:Button label="+" click="addEmployee()"/>
       <mx:Button Label="-"
         click="company.employees.removeItemAt
         (dgEmployees.selectedIndex)"/>
     </mx:HBox>
     <mx:DataGrid id="dgEmployees" dataProvider="{company.employees}"
       width="100%" height="100%" editable="true">
       <mx:columns>
         <mx:DataGridColumn dataField="firstName"
           headerText="First Name"/>
         <mx:DataGridColumn dataField="lastName" headerText="Last Name"/>
         <mx:DataGridColumn dataField="title" headerText="Title"/>
         <mx:DataGridColumn dataField="email" headerText="Email"/>
         <mx:DataGridColumn dataField="phone" headerText="Phone"/>
       </mx:columns>
     </mx:DataGrid>
   </mx:VBox>
</mx:TabNavigator>
```

Assembler and DAO code

When you implement your assembler, you have two options for how data in your relationships is returned. If you use lazy associations, the parent assembler can return instances of the child objects that have only the IDs populated. If your association is not lazy, it must return fully populated instances. This example demonstrates a variant of the `getCompany()` method that returns a fully populated collection of employees. Notice that in this example, a company_employee join table is used where the company owns the employees property and there is no company property in the employee.

In a more traditional schema, you might store the company_id in the employee table and employee lists would be obtained by fill methods because they might be large. If you implemented the schema that way, your employee destination would have a company association and it would be responsible for saving the value of the company_id column.

You can also have associations in both directions. In this case, the client code has to update both sides of the relationship; for example, you would add the employee to the `company.employees` property and set the company in the `employee.company` property. Your assembler must choose an owner destination for the association property that indicates which destination is responsible for updating the changes. In this case, you would usually ignore changes to the `company.employees` property. You can mark it as read-only by using the `read-only` attribute.

The following example shows the `getCompany()` methods of the CompanyAssembler class:

```
  public Company getCompany(Map uid)
  {
    CompanyDAO dao = new CompanyDAO();
    return dao.getCompany(((Integer) uid.get("companyId")).intValue());
  }
```

The `getCompany()` method is invoked from the client when a user clicks on a row in the DataGrid of company summary data. The `getCompany()` method calls the `CompanyDAO.getCompany()` method. The `CompanyDAO.getCompany()` method does a SQL query from the company database table to retrieve the company information. It then does a separate query that joins the company_employee and employee tables to retrieve the employees for that company. The following example shows the `CompanyDAO.getCompany()` method:

```
// CompanyDAO.java
...
  public Company getCompany(int companyId) throws DAOException
  {
    Company company = null;
    Connection c = null;
    try
    {
      c = ConnectionHelper.getConnection();
      Statement s = c.createStatement();
      ResultSet rs = s.executeQuery("SELECT * FROM company WHERE
        company_id=" + companyId);
      if (rs.next())
      {
        company = new Company();
        company.setCompanyId(rs.getInt("company_id"));
        company.setName(rs.getString("name"));
        company.setAddress(rs.getString("address"));
        company.setCity(rs.getString("city"));
```

```
        company.setZip(rs.getString("zip"));
        company.setState(rs.getString("state"));
        List employees = new ArrayList();
        rs = s.executeQuery("SELECT a.employee_id, first_name,
          last_name, title, email, phone FROM company_employee AS a,
          employee AS b WHERE a.employee_id=b.employee_id AND
          company_id=" + companyId);
        while (rs.next())
        {
          employees.add(new Employee(
               rs.getInt("employee_id"),
               rs.getString("first_name"),
               rs.getString("last_name"),
               rs.getString("title"),
               rs.getString("email"),
               rs.getString("phone")));
        }
        company.setEmployees(employees);
      }
    }
```

The company destination and employee destination each have a sync method that handles server-side data synchronization of company or employee data, respectively. The sync methods are no different than sync methods for destinations that do not have a nested relationship. However, when a data conflict occurs for either destination, the code uses the same conflict handler to ensure conflicts are processed in the proper order in case there are dependencies between them. Depending on the type of change, the sync method calls the assembler's `doCreate()`, `doUpdate()`, or `doDelete()` method. The `doCreate()` method is responsible for persisting the initial set of the IDs of any associated objects for any associations it owns. The `doUpdate()` method is responsible for updating the list of IDs given the old and new versions of the object. It is able to detect conflicts on the associated IDs. The `doDelete()` method is responsible for deleting the IDs of any associated objects for associating which it owns.

Implementing an object relationship in a fill method

When you use the Java adapter, you can implement a relationship between two objects by using queries where one of the parameters of a fill method defined in the assembler is the ID of the related object. This query approach is more efficient for large collections of objects than the managed association approach; for example, you have a Company object that has a large number of Employee instances.

The query approach to implementing a relationship has some characteristics that make it not appropriate for all situations. When you change an employee's `companyId`, the Data Management Service must update the results of the fill methods affected by that change for clients that display the list of employees to be updated automatically. By default, it uses the auto-refresh feature to re-execute every outstanding fill method that returns employees. You can tune this to avoid re-executing these queries so aggressively, but that requires some additional coding in your assembler. For more information about using auto-refresh, see "Detecting changes to fill method results" on page 1531.

The query approach also does not detect conflicts when you update that association. You would not want to detect a conflict if two clients added an employee to the same company at roughly the same time, but you might want to detect a conflict if two clients simultaneously updated the same customer's list of addresses. With managed association properties, you can use the Data Management Service's conflict detection mechanism to detect data conflicts on the complete contents of the collections; however, with a fill method, conflicts are only detected on the properties of the item, not the filled collection the item was returned in.

The example code in this section is from the CRM application, which is included in the samples web application included with Flex Data Services. This application implements the relationship between companies and employees in the `fill()` method of the EmployeeAssembler class. The source code for the application's Java assembler and DAO classes are in the WEB_INF/src/samples/crm directory of the samples web application.

The following example shows the `fill()` method of the EmployeeAssembler class in the CRM application. The boldface text highlights the part of the method that finds employees by company, based on the numeric company ID provided in a client's fill request.

```
...
public Collection fill(List fillParameters)
{
  EmployeeDAO dao = new EmployeeDAO();
    if (fillParameters.size() == 0)
      return dao.getEmployees();

  String queryName = (String) fillParameters.get(0);
  if (queryName.equals("match"))
    return dao.findMatchingEmployees((String) fillParameters.get(1));
  if (queryName.equals("byCompany"))
    return dao.findEmployeesByCompany((Integer)
      fillParameters.get(1));
  return super.fill(fillParameters); // throws a nice error
}
```

The boldface code in the following examples is the corresponding client-side fill request that gets employees by company:

```
private function companyChange():void {
  if (dg.selectedIndex > -1)
  {
    if (company != companies.getItemAt(dg.selectedIndex))
    {
      company = Company(companies.getItemAt(dg.selectedIndex));
      dsEmployee.fill(employees, "byCompany",
      company.companyId);
    }
  }
}
```

Pushing data changes from the server to clients

You use the flex.data.DataServiceTransaction class to push server-side data changes to clients. This class is documented in the public Flex Data Services Javadoc documentation. An instance of the DataServiceTransaction class is created for each operation that modifies the state of objects that the Data Management Service manages. You can use this object from server-side code to push changes to managed data stored on clients that have the `autoSyncEnabled` property of your DataService component set to `true`.

If you add changes from within a sync method, the Data Management Service creates an instance of the DataServiceTransaction class, so you can call its static `getCurrentDataServiceTransaction()` method and then call the `updateItem()`, `deleteItem()`, and `createItem()` methods to trigger additional changes. You should call these methods to apply changes you have persisted or will be persisting in this transaction. If the current transaction is rolled back, these changes are not pushed to clients.

To roll back a transaction, you mark the javax.transaction.UserTransaction instance as rolled back as you would in a normal J2EE application or you can call the `setRollbackOnly()` method on the DataServiceTransaction.

The DataServiceTransaction class provides access to the current transaction in a thread local state, and holds messages to be pushed to clients when the transaction completes. You also use this class to register for synchronization events before and after completion of the transaction.

Each DataServiceTransaction instance is stored in thread-local state and it is assumed that it is only operated on one thread at a time. It is not thread safe.

If you use the DataServiceTransaction class from within a sync, update, create, or delete method of your assembler, do not use this class to indicate changes made to items that are already in the midst of being changed in this transaction. Doing so queues an additional change to that object instead of modifying the currently active one, which can create a conflict. Instead, you update the NewVersion instance with your changed property values and add any newly changed property values to the list of changed properties sent to you. For example, if after every update made to an instance, you want to change the `versionId` of that instance, you can add the `versionId` to the list of changes and also update the `versionId` value in your newVersion instance.

If you are using the ChangeObject interface, you call the `addChangedPropertyName()` method to add the `versionId` property. If you are using the `updateItem()` method, you just add that property to the list provided to your `updateItem()` method.

The `DataServiceTransaction.refreshFill()` method lets you manually refresh a fill or matching set of fills from server code either as part of your assembler's sync method or from other server-side code. You specify a list of fill parameters that are used to create a matching list of fills that are currently being cached by active clients. This list can be null, which means that all fills on that destination match. If the list is non-null, the `refreshFill()` matches fills made by clients with the same number of parameters if all of the slots match based on the rules in the following table:

Value	Rule
Null value	Matches that slot unconditionally.
Class value	Matches a parameter in that slot of that type.
Any other value	Matches fill parameters by using the equals method.

NOTE

When you compile code that uses Flex Data Services Java APIs, you must include the messaging.jar and flex-messaging-common.jar files in your classpath.

PART 7

Charting Components

7

This part describes how to use charting components with Adobe Flex 2.

The following topics are included:

CHAPTER 53

Introduction to Charts

53

Displaying data in a chart or graph can make data interpretation much easier for users of the applications that you develop with the Adobe Flex 2 product line. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other type of chart using colors, captions, and a two-dimensional representation of your data.

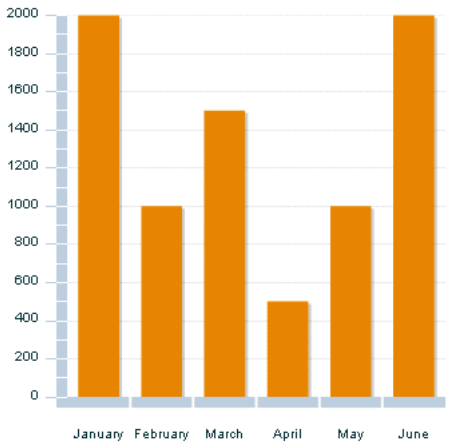
This topic introduces you to the concepts for using the Adobe Flex Charting controls.

Contents

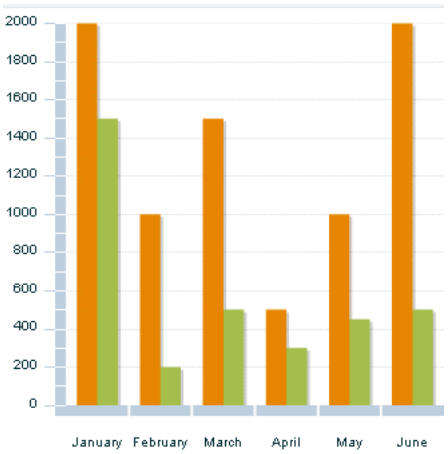
About charting

Data visualization lets you present data in a way that simplifies data interpretation and data relationships. Charting is one type of data visualization in which you create two-dimensional representations of your data. Flex supports some of the most common types of two-dimensional charts (such as bar, column, and pie charts) and gives you a great deal of control over the appearance of charts.

A simple chart shows a single data series, where a series is a group of related data points. For example, a data series might be monthly sales revenues or daily occupancy rates for a hotel. The following chart shows a single data series that corresponds to sales over several months.



Another chart might add a second data series. For example, you might include the percentage growth of profits over the same four business quarters. The following chart shows two data series—one for sales and one for profit.



For information about installing Flex Charting, see the readme.txt file included with the Flex Charting installer.

Using the charting controls

Flex Charting lets you create some of the most common chart types, and also lets you customize the appearance of your charts. The charting controls are located in the mx.charts.* package.

The following table lists the supported chart types, the name of the control class, and the name of the series class that you use to define what data appears in each chart.

Chart type	Chart control class	Chart series class
Area	AreaChart	AreaSeries
Bar	BarChart	BarSeries
Bubble	BubbleChart	BubbleSeries
Candlestick	CandlestickChart	CandlestickSeries
Column	ColumnChart	ColumnSeries
HighLowOpenClose	HLOCChart	HLOCSeries
Line	LineChart	LineSeries
Pie	PieChart	PieSeries
Plot	PlotChart	PlotSeries

All chart controls, except the PieChart class, are subclasses of the CartesianChart class. Cartesian charts are charts that typically represent a set of data points in rectangular-shaped, two-dimensional space. The PieChart class is a subclass of the PolarChart class, which represents data in circular space.

All chart controls inherit basic charting characteristics from the ChartBase class.

A chart control typically has the following structure in MXML:

```
<mx:ChartName>
  <!-- Define one or more series. -->
  <mx:SeriesName/>
  <mx:SecondSeriesName/>

  <!-- Define the axes. -->
  <mx:horizontalAxis>
    <mx:AxisType/>
  </mx:horizontalAxis>
  <mx:verticalAxis>
    <mx:AxisType/>
  </mx:verticalAxis>

  <!-- Style the axes and ticks. -->
  <mx:horizontalAxisRenderer>
```

```
    <mx:AxisRenderer/>
  </mx:horizontalAxisRenderer>
  <mx:verticalAxisRenderer>
    <mx:AxisRenderer/>
  </mx:verticalAxisRenderer/>

  <!-- Add grid lines and other elements to the chart. -->
  <mx:annotationElements>
    <mx:Array/>
  </mx:annotationElements>
  <mx:backgroundElements>
    <mx:Array/>
  </mx:backgroundElements/>
</mx:ChartName>

<!-- Optionally define the legend. -->
<mx:Legend/>
```

The following table describes the parts of the chart in more detail:

Part	Description
Chart	(Required) Defines one or two data providers for the chart. Also defines the chart type and sets data tips, mouse sensitivity, gutter styles, and axis styles. This is the top-level tag for a chart control. All other tags are child tags of this tag.
Series	(Required) Defines one or more data series to be displayed on the chart. Also sets the strokes, fills, and renderers (or *skins*) of the data series, as well as the strokes and fills used by the chart's legend for each series. You can also define a second set of series for each chart, to show multiple data series in a single chart. Each series in a chart can have its own data provider.
Axes	Sets the axis type (numeric or category). Also defines the axis labels, titles, and style properties such as padding. You can also define axes for the second set of series, if there is one.
Axes renderer	(Optional) Sets tick placement and styles, enables or disables labels, and defines axis lines, label rotation, and label gap. You can also define an axis renderer for a second series, if there is one.
Elements	(Optional) Defines grid lines and extra elements to appear on the chart.

For each chart type, Flex supplies a corresponding chart control and chart series. The chart control defines the chart type, the data provider that supplies the chart data, the grid lines, the text for the chart axes, and other properties specific to the chart type. The `dataProvider` property of the chart control determines what data the chart uses.

A *data provider* is a collection of objects. It can be an Array of objects or any object that implements the collections API. A data provider can also be an XMLList object with XML nodes, such as the result of an E4X query.

The chart components use a flat, or list-based, data provider similar to a one-dimensional array. The data provider can contain objects such as Strings and Numbers, or even other objects. For more information on supplying chart data, see "Defining chart data" on page 1592.

You use the chart series to identify which data from the data provider the chart displays. A data provider can contain more data than you want to show in your chart, so you use the chart's series to specify which points you want to use from the data provider. You can specify a single data series or a second series. You can also use the chart series to define the appearance of the data in the chart.

All chart series inherit the data provider from the chart unless they have a data provider explicitly set on themselves. If you set the value of the `dataProvider` property on the chart control, you are not required to set the property value on the series. You can, however, define different data providers for each series in a chart.

For example, to create a pie chart, you use the PieChart control with the PieSeries chart series. To create an area chart, you use the AreaChart control with the AreaSeries chart series, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicAreaOneSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

This example defines an array containing a single `<mx:AreaSeries>` tag. The `<mx:AreaSeries>` tag specifies the single data series that is displayed in the chart.

You can add a second <mx:AreaSeries> tag to display two data series, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You are not required to define a data provider on the chart control. Each series can have its own data provider, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/MultipleDataProviders.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var profit04:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000},
        {Month:"Feb", Profit:1000},
        {Month:"Mar", Profit:1500}
     ]);
     [Bindable]
     public var profit05:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2200},
        {Month:"Feb", Profit:1200},
        {Month:"Mar", Profit:1700}
     ]);
     [Bindable]
     public var profit06:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2400},
        {Month:"Feb", Profit:1400},
        {Month:"Mar", Profit:1900}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{profit04}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                dataProvider="{profit04}"
                yField="Profit"
                xField="Month"
                displayName="2004"
           />
           <mx:ColumnSeries
                dataProvider="{profit05}"
                yField="Profit"
                xField="Month"
                displayName="2005"
           />
           <mx:ColumnSeries
                dataProvider="{profit06}"
                yField="Profit"
                xField="Month"
                displayName="2006"
```

```
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can use the `secondDataProvider`, `secondSeries`, `secondVerticalAxis`/`secondHorizontalAxis`, or `secondVerticalAxisRenderer`/`secondHorizontalAxisRenderer` properties to add a second data provider, series, axis, or axis renderer to your chart. For more information, see "Using multiple data series" on page 1671 and "Using multiple axes" on page 1673.

To dynamically size the chart to the size of the browser window, set the `width` and `height` attributes to a percentage value, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicBarSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart" height="1000" width="1000">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        height="100%"
        width="100%"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Parent containers of the charting control must also be set using percentage values; otherwise, the chart does not resize when the window resizes.

About the axes

Chart data is defined by the fields specified in the chart's data series. The appearance and contents of *axis labels* are defined by the `<mx:horizontalAxis>` (x-axis) and `<mx:verticalAxis>` (y-axis) tags and the renderers for these tags (`<mx:horizontalAxisRenderer>` and `<mx:verticalAxisRenderer>`). These tags not only define the data ranges that appear in the chart, but also map the data points to their names and labels. This latter mapping has a large impact on how the Data Management Service chart renders the values of DataTip labels, axis labels, and tick marks.

Flex Charting supports the following types of axes:

CategoryAxis A CategoryAxis class maps a set of values (such as stock ticker symbols, state names, or demographic categories) to the axis. You use the `<mx:CategoryAxis>` tag to define axis labels that are grouped by logical associations and that are not necessarily numeric. For example, the month names used in the chart in "About charting" on page 1573 could be defined as a CategoryAxis class.

LinearAxis A LinearAxis class maps numeric data to the axis. You use the `<mx:LinearAxis>` child tag of the `<mx:horizontalAxis>` or `<mx:verticalAxis>` tags to customize the range of values displayed along the axis, and to set the increment between the axis labels of the tick marks.

LogAxis A LogAxis class maps numeric data to the axis logarithmically. You use the `<mx:LogAxis>` child tag of the `<mx:horizontalAxis>` or `<mx:verticalAxis>` tags. Labels on the logarithmic axis are even powers of 10.

DateTimeAxis A DateTimeAxis class maps time-based values, such as hours, days, weeks, or years, along a chart axis. You use the `<mx:DateTimeAxis>` tag to define the axis labels.

The DateTimeAxis, LogAxis, and LinearAxis are all of type NumericAxis, because they are used to represent numeric values. In many cases, you are required to define only one axis as being a NumericAxis or a CategoryAxis. Flex assumes that all axes not explicitly defined are of type LinearAxis. However, to use decorations such as DataTip labels and legends, you might be required to explicitly define both axes.

For a PlotChart control, both axes are considered a LinearAxis, because the data point is the intersection of two coordinates. So, you are not required to specify either axis, although you can do so to provide additional settings, such as minimum and maximum values.

Each axis can have a corresponding AxisRenderer object (specified by the `horizontalAxisRenderer` or `verticalAxisRenderer` properties) that defines the appearance of axis labels and tick marks. In addition to defining formats, you can use an AxisRenderer class to customize the value of the axis labels. For more information, see Chapter 55, "Formatting Charts," on page 1681.

By default, Flex uses the chart type and the orientation to calculate the labels that appear along the x-axis and y-axis of the chart. A column chart, for example, has the following default values:

The x-axis The minimum value is 0, and the maximum value is the number of items in the data series that is being charted.

The y-axis Flex calculates the minimum value on the y-axis to be small enough for the chart data, and the maximum value to be large enough based on the chart data.

For more information about chart axes, see "Working with axes" on page 1698.

About charting events

The chart controls include events that accommodate user interaction with data points in charts. These events are described in Chapter 56, "Using Events and Effects in Charts," on page 1801.

Creating charts in ActionScript

You can create, destroy, and manipulate charts using ActionScript just as you can any other Flex component.

When working in Script blocks or in separate ActionScript class files, you must be sure to import all appropriate classes. The following set of import statements defines the most common cases:

```
import mx.collections.*;
import mx.charts.*;
import mx.charts.series.*;
import mx.charts.renderers.*;
import mx.charts.events.*;
```

To create a chart in ActionScript, use the `new` keyword. You can set properties on the chart object as you would in MXML. You assign a data provider with the `dataProvider` property. To add a data series to the chart, you define a new data series of the appropriate type. To apply the series to your chart, use the chart's `series` property. You can specify the category axis settings using the CategoryAxis class. The following example defines a BarChart control with two series:

```
<?xml version="1.0"?>
<!-- charts/CreateChartInActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.BarChart;
     import mx.charts.series.BarSeries;
     import mx.charts.CategoryAxis;
     import mx.charts.Legend;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     public var myChart:BarChart;
     public var series1:BarSeries;
     public var series2:BarSeries;
     public var legend1:Legend;

     public function init():void {
        // Create the chart object and set some
        // basic properties.
        myChart = new BarChart();
        myChart.showDataTips = true;
        myChart.dataProvider = expenses;

        // Define the category axis.
        var vAxis:CategoryAxis = new CategoryAxis();
        vAxis.categoryField = "Month" ;
        vAxis.dataProvider =  expenses;
        myChart.verticalAxis = vAxis;

        // Add the series.
        var mySeries:Array=new Array();
        series1 = new BarSeries();
        series1.xField="Profit";
        series1.yField="Month";
        series1.displayName = "Profit";
```

```
        mySeries.push(series1);

        series2 = new BarSeries();
        series2.xField="Expenses";
        series2.yField="Month";
        series2.displayName = "Expenses";
        mySeries.push(series2);

        myChart.series = mySeries;

        // Create a legend.
        legend1 = new Legend();
        legend1.dataProvider = myChart;

        // Attach chart and legend to the display list.
        p1.addChild(myChart);
        p1.addChild(legend1);
    }
  ]]></mx:Script>
  <mx:Panel id="p1" title="Bar Chart Created in ActionScript"/>
</mx:Application>
```

This example replaces the existing Array of series with the new series.

You can use a similar technique to add data series to your charts rather than replacing the existing ones. The following example creates two ColumnSeries and sets their data providers. It then creates an Array that holds the existing chart series, and pushes the new series into that Array. Finally, it sets the value of the chart's `series` property to be the new Array of series.

```
<?xml version="1.0"?>
<!-- charts/AddingSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.series.ColumnSeries;

     [Bindable]
     public var profit04:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2000},
        {Month: "Feb", Profit: 1000},
        {Month: "Mar", Profit: 1500}
     ]);
     [Bindable]
     public var profit05:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2200},
        {Month: "Feb", Profit: 1200},
        {Month: "Mar", Profit: 1700}
     ]);
     [Bindable]
     public var profit06:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2400},
        {Month: "Feb", Profit: 1400},
        {Month: "Mar", Profit: 1900}
     ]);

     public var series1:ColumnSeries;
     public var series2:ColumnSeries;

     public function addMoreSeries():void {
        if (!series1 || !series2) {
           series1 = new ColumnSeries();
           series1.dataProvider = profit05;
           series1.yField = "Profit";
           series1.xField = "Month";
           series1.displayName = "2005";

           series2 = new ColumnSeries();
           series2.dataProvider = profit06;
           series2.yField = "Profit";
           series2.xField = "Month";
           series2.displayName = "2006";

           var currentSeries:Array = myChart.series;
```

```
            currentSeries.push(series1);
            currentSeries.push(series2);

            myChart.series = currentSeries;
        }
    }
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{profit04}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries dataProvider="{profit04}"
                yField="Profit"
                xField="Month"
                displayName="2004"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
  <mx:Button id="b1" label="Add More Series To Chart" click=
  "addMoreSeries()"/>
</mx:Application>
```

By using ActionScript, you can also define a variable number of series for your charts. The following example uses E4X syntax to extract an Array of unique names from the data. It then iterates over this Array and builds a new LineSeries for each name.

```
<?xml version="1.0"?>
<!-- charts/VariableSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp();">
  <mx:Script><![CDATA[
        import mx.charts.series.LineSeries;
        import mx.charts.DateTimeAxis;

        [Bindable]
        private var myXML:XML =
            <dataset>
            <item>
                <who>Tom</who>
                <when>08/22/2006</when>
                <hours>5.5</hours>
            </item>
            <item>
                <who>Tom</who>
                <when>08/23/2006</when>
                <hours>6</hours>
            </item>
```

```
    <item>
        <who>Tom</who>
        <when>08/24/2006</when>
        <hours>4.75</hours>
    </item>
    <item>
        <who>Dick</who>
        <when>08/22/2006</when>
        <hours>6</hours>
    </item>
    <item>
        <who>Dick</who>
        <when>08/23/2006</when>
        <hours>8</hours>
    </item>
    <item>
        <who>Dick</who>
        <when>08/24/2006</when>
        <hours>7.25</hours>
    </item>
    <item>
        <who>Jane</who>
        <when>08/22/2006</when>
        <hours>6.5</hours>
    </item>
    <item>
        <who>Jane</who>
        <when>08/23/2006</when>
        <hours>9</hours>
    </item>
    <item>
        <who>Jane</who>
        <when>08/24/2006</when>
        <hours>3.75</hours>
    </item>
    </dataset>;

public function initApp():void {
    var wholist:Array = new Array();
    for each(var property:XML in myXML.item.who) {
        // Create an Array of unique names.
        if (wholist[property] != property)
            wholist[property] = property;
    }

    // Iterate over names and create a new series
    // for each one.
    for (var s:String in wholist) {
        // Use all items whose name matches s.
        var localXML:XMLList = myXML.item.(who==s);
```

```
                // Create the new series and set its properties.
                var localSeries:LineSeries = new LineSeries();
                localSeries.dataProvider = localXML;
                localSeries.yField = "hours";
                localSeries.xField = "when";

                // Set values that show up in dataTips and Legend.
                localSeries.displayName = s;

                // Back up the current series on the chart.
                var currentSeries:Array = myChart.series;
                // Add the new series to the current Array of series.
                currentSeries.push(localSeries);
                // Add the new Array of series to the chart.
                myChart.series = currentSeries;
            }

            // Create a DateTimeAxis horizontal axis.
            var hAxis:DateTimeAxis = new DateTimeAxis();
            hAxis.dataUnits = "days";
            // Set this to false to display the leftmost label.
            hAxis.alignLabelsToUnits = false;
            // Take the date in its current format and create a Date
            // object from it.
            hAxis.parseFunction = createDate;
            myChart.horizontalAxis = hAxis;
        }

        public function createDate(s:String):Date {
            // Reformat the date input to create Date objects
            // for the axis.
            var a:Array = s.split("/");

            // The existing String s is in the format "MM/DD/YYYY".
            // To create a Data object, you pass "YYYY,MM,DD",
            // where MM is zero-based, to the Date() constructor.
            var newDate:Date = new Date(a[2],a[0]-1,a[1]);
            return newDate;
        }
    ]]></mx:Script>

  <mx:Panel title="Line Chart with Variable Number of Series">
     <mx:LineChart id="myChart" showDataTips="true"/>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Defining chart data

The chart controls have a `dataProvider` property that defines the data for the chart. The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple charts from the same data provider, switch data providers for a chart at run time, and modify the data provider so that changes are reflected by all charts using the data provider.

Using chart data

To use the data from a data provider in your chart control, you map the `xField` and `yField` properties of the chart series to the fields in the data provider. The `xField` property defines the data for the horizontal axis, and the `yField` property defines the data for the vertical axis.

For example, assume your data provider has the following structure:

```
{Month: "Feb", Profit: 1000, Expenses: 200, Amount: 60}
```

You can use the `Profit` and `Expenses` fields and ignore the `Month` field by mapping the `xField` property of the series object to one field and the `yField` property of the series object to another field, as the following example shows:

```
<mx:PlotSeries xField="Profit" yField="Expenses"/>
```

The result is that each data point is the intersection of the `Profit` and `Expenses` fields from the data provider.

To place the data points into a meaningful grouping, you can choose a separate property of the data provider as the `categoryField`. In this case, to sort each data point by month, you map the `Month` field to the `categoryField` property of the horizontal axis:

```
<mx:horizontalAxis>
  <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month"/>
</mx:horizontalAxis>
```

In some cases, depending on the type of chart and the type of data you are representing, you use either the `xField` property or the `yField` property to define the data series. In a ColumnChart control, for example, the `yField` property defines the height of the column. You do not have to specify an `xField` property. To get an axis label for each column, you specify a `categoryField` property for the `horizontalAxis`.

When you use chart data, keep the following in mind:

- You usually match a series with a data provider field if you want to display that series. However, this is not always true. If you do not specify an `xField` for a ColumnSeries, Flex assumes the index is the value. If you do not specify a `yField`, Flex assumes the data provider is a collection of *y* values, rather than a collection of objects that have *y* values. For example, the following series renders correctly for a ColumnChart control:
  ```
  <mx:ColumnSeries dataProvider="{[1,2,3,4,5]}"/>
  ```
- Some series use only one field from the data provider, while others can use two or more. For example, you specify only a `field` property for a PieSeries object, but you can specify an `xField` and a `yField` for a PlotSeries object and an `xField`, `yField`, and `radiusField` for a BubbleSeries object.
- Most of the series can determine suitable defaults for their nonprimary dimensions if no field is specified. For example, if you do not explicitly set an `xField` for the ColumnSeries, LineSeries, and AreaSeries, Flex maps the data to the chart's categories in the order in which the data appears in the data provider. Similarly, a BarSeries maps the data to the categories if you do not set a `yField`.

For a complete list of the fields that each data series can take, see the data series' entry in *Adobe Flex 2 Language Reference*. For more information on data providers, see "Using data provider controls" on page 262.

Types of chart data

You can supply data to a data provider in the following ways:

- Define it in a `<mx:Script>` block.
- Define it in an external XML, ActionScript, or text file.
- Return it by using a WebService call.
- Return it by using a RemoteObject component.
- Return it by using an HTTPService component.
- Define it in MXML.

For more information on data providers, see Chapter 7, "Using Data Providers and Collections," on page 161.

Using static Arrays as data providers

Using a static Array of objects for the data provider is the simplest approach. You typically create an Array of objects, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfObjectsDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     private var expenses:Array = [
        {Month:"January",Profit:2000,Expenses:1500,Amount:450},
        {Month:"February",Profit:1000,Expenses:200,Amount:600},
        {Month:"March",Profit:1500,Expenses:500,Amount:300},
        {Month:"April",Profit:500,Expenses:300,Amount:500},
        {Month:"May",Profit:1000,Expenses:450,Amount:250},
        {Month:"June",Profit:2000,Expenses:500,Amount:700}
     ];

  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use MXML to define the content of an Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfMXMLObjectsDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Array id="expenses">
     <mx:Object
        Month="January"
        Profit="2000"
        Expenses="1500"
        Amount="450"
     />
     <mx:Object
        Month="February"
        Profit="1000"
        Expenses="200"
        Amount="600"
     />
     <mx:Object
        Month="March"
        Profit="1500"
        Expenses="500"
        Amount="300"
     />
     <mx:Object
        Month="April"
        Profit="500"
        Expenses="300"
        Amount="500"
     />
     <mx:Object
        Month="May"
        Profit="1000"
        Expenses="450"
        Amount="250"
     />
     <mx:Object
        Month="June"
        Profit="2000"
        Expenses="500"
        Amount="700"
     />
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
```

```
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also define objects in MXML with a more verbose syntax, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfVerboseMXMLObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Array id="expenses">
     <mx:Object>
       <mx:Month>January</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>1500</mx:Expenses>
       <mx:Amount>450</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>February</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>200</mx:Expenses>
       <mx:Amount>600</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>March</mx:Month>
       <mx:Profit>1500</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>April</mx:Month>
       <mx:Profit>500</mx:Profit>
       <mx:Expenses>300</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>May</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>450</mx:Expenses>
       <mx:Amount>250</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>June</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>700</mx:Amount>
     </mx:Object>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
```

```
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

A disadvantage of using a simple Array as a chart's data provider is that you can use only the methods of the Array class to manipulate the data. In addition, when you use an Array as a data provider, the data in it must be static. Even if you make the Array bindable, when data in an Array changes, the chart does not reflect those changes. For more robust data manipulation and data binding, you can use a collection for the chart data provider, as described in "Using collections as data providers" on page 1598.

Using collections as data providers

Collections are a more robust data provider mechanism than Arrays. They provide operations that include the insertion and deletion of objects as well as sorting and filtering. Collections also support change notification. An ArrayCollection object provides an easy way to expose an Array as an ICollectionView or IList interface.

As with Arrays, you can use MXML to define the contents of a collection, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfMXMLObjectsDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:ArrayCollection id="expenses">
     <mx:Object
        Month="January"
        Profit="2000"
        Expenses="1500"
        Amount="450"
     />
     <mx:Object
        Month="February"
        Profit="1000"
        Expenses="200"
        Amount="600"
     />
     <mx:Object
        Month="March"
        Profit="1500"
        Expenses="500"
        Amount="300"
     />
     <mx:Object
        Month="April"
        Profit="500"
        Expenses="300"
        Amount="500"
     />
     <mx:Object
        Month="May"
        Profit="1000"
        Expenses="450"
        Amount="250"
     />
     <mx:Object
        Month="June"
        Profit="2000"
        Expenses="500"
        Amount="700"
     />
  </mx:ArrayCollection>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
```

```
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Or you can define a complex object in MXML:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfVerboseMXMLObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:ArrayCollection id="expenses">
     <mx:Object>
       <mx:Month>January</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>1500</mx:Expenses>
       <mx:Amount>450</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>February</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>200</mx:Expenses>
       <mx:Amount>600</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>March</mx:Month>
       <mx:Profit>1500</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>April</mx:Month>
       <mx:Profit>500</mx:Profit>
       <mx:Expenses>300</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>May</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>450</mx:Expenses>
       <mx:Amount>250</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>June</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>700</mx:Amount>
     </mx:Object>
  </mx:ArrayCollection>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
```

```
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can create an ArrayCollection object in ActionScript. If you define an ArrayCollection in this way, ensure that you import the mx.collections.ArrayCollection class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     private var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If your data is in an Array, you can pass the Array to the ArrayCollection's constructor to convert it to an ArrayCollection. The following example creates an Array, and then converts it to an ArrayCollection:

```
<?xml version="1.0"?>
<!-- charts/ArrayConvertedToArrayCollection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     private var expenses:Array = [
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ];

     [Bindable]
     public var expensesAC:ArrayCollection =
        new ArrayCollection(expenses);

  ]]></mx:Script>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expensesAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expensesAC}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Similarly, you can use an <mx:ArrayCollection> tag to perform the conversion:

```
<?xml version="1.0"?>
<!-- charts/ArrayConvertedToArrayCollectionMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     private var expenses:Array = [
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ];
  ]]></mx:Script>

  <mx:ArrayCollection id="expensesAC" source="{expenses}"/>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expensesAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expensesAC}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The data in ArrayCollections can be bound to the chart's data provider so that the data can be updated in real-time. The following example creates an object with elapsed time and total memory usage every second. It then pushes that new object onto an ArrayCollection that is used as the data provider for a line chart. As a result, the chart itself updates every second showing memory usage of Flash Player over time.

```
<?xml version="1.0"?>
<!-- charts/RealTimeArrayCollection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"initTimer()">
  <mx:Script><![CDATA[
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  import mx.collections.ArrayCollection;

  [Bindable]
  public var memoryUsage:ArrayCollection = new ArrayCollection();

  public function initTimer():void {
     // The first parameter in the Timer constructor
     // is the interval, in milliseconds.
     // The second parameter is how many times to run (0 is
     // infinity).
     var myTimer:Timer = new Timer(1000, 0);

     // Add the listener for the timer event.
     myTimer.addEventListener("timer", timerHandler);
     myTimer.start();
  }

  public function timerHandler(event:TimerEvent):void {
     var o:Object = new Object();

     // Get the number of milliseconds since Flash Player started.
     o.time = getTimer();

     // Get the total memory Flash Player is using.
     o.memory = flash.system.System.totalMemory;
     trace(o.time + ":" + o.memory);

     // Add new object to the ArrayCollection, which is bound
     // to the chart's data provider.
     memoryUsage.addItem(o);
  }
  ]]></mx:Script>

  <mx:LineChart id="chart" dataProvider="{memoryUsage}"
     showDataTips="true">
     <mx:horizontalAxis>
```

```
        <mx:LinearAxis/>
      </mx:horizontalAxis>
      <mx:verticalAxis>
         <mx:LinearAxis minimum="5000000"/>
      </mx:verticalAxis>
      <mx:series>
         <mx:LineSeries yField="memory"/>
      </mx:series>
   </mx:LineChart>
</mx:Application>
```

Data collections can be paged, which means that data is sent to the client in chunks as the application requests it. But Flex Charting components display all of the data all of the time, by default. As a result, when you use data collections with charts, you should disable the paging features or use non-paged views of the data collection for chart data. For more information on using collections, see "About collections" on page 167.

Using an XML file as a data provider

You can define data provider data in a structured file. The following example shows the contents of the data.xml file:

```
<data>
<result month="Jan-04">
<apple>81768</apple>
<orange>60310</orange>
<banana>43357</banana>
</result>
<result month="Feb-04">
<apple>81156</apple>
<orange>58883</orange>
<banana>49280</banana>
</result>
</data>
```

You can load the file directly as a source of a Model, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/XMLFileDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Model id="results" source="../assets/data.xml"/>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{results.result}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="banana" displayName="Banana"/>
           <mx:LineSeries yField="apple" displayName="Apple"/>
           <mx:LineSeries yField="orange" displayName="Orange"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

To use an ArrayCollection as the chart's data provider, you convert the Model to one, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/XMLFileToArrayCollectionDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%">
  <mx:Script>
     import mx.utils.ArrayUtil;
  </mx:Script>

  <mx:Model id="results" source="../assets/data.xml"/>
  <mx:ArrayCollection id="myAC"
        source="{ArrayUtil.toArray(results.result)}"
  />

  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{myAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="banana" displayName="Banana"/>
           <mx:LineSeries yField="apple" displayName="Apple"/>
           <mx:LineSeries yField="orange" displayName="Orange"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also define the XML file as a URL for an HTTPService component, and then bind the HTTPService result directly to the chart's data provider, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%" creationComplete="srv.send()">
  <mx:HTTPService id="srv" url="../assets/data.xml"/>

  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart"
        dataProvider="{srv.lastResult.data.result}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="apple" name="Apple"/>
           <mx:LineSeries yField="orange" name="Orange"/>
           <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

To use an ArrayCollection, you convert the HTTPService result to an ArrayCollection, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceToArrayCollectionDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="srv.send()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var myData:ArrayCollection;
  ]]></mx:Script>

  <mx:HTTPService
     id="srv"
     url="../assets/data.xml"
     useProxy="false"
     result="myData=ArrayCollection(srv.lastResult.data.result)"
  />
  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{myData}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="apple" name="Apple"/>
           <mx:LineSeries yField="orange" name="Orange"/>
           <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also set the result format of the HTTPService to E4X, and then use it as a source for an XMLListCollection object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceToXMLListCollection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="srv.send()">
  <mx:Script><![CDATA[
     import mx.utils.ArrayUtil;
  ]]></mx:Script>

  <mx:HTTPService id="srv"
    url="../assets/data.xml"
    resultFormat="e4x"
  />

  <mx:XMLListCollection id="myAC"
    source="{srv.lastResult.result}"
  />

  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{myAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="apple" name="Apple"/>
           <mx:LineSeries yField="orange" name="Orange"/>
           <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

Randomly generating chart data

A useful way to create data for use in sample charts is to generate random data. The following example generates test data for use with the chart controls:

```
<?xml version="1.0"?>
<!-- charts/RandomDataGeneration.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.collections.*;

     // Define data provider array for the chart data.
     [Bindable]
     public var dataSet:ArrayCollection;

     // Define the number of elements in the array.
     public var dsLength:Number = 10;

     public function initApp():void {
        // Initialize data provider array.
        dataSet = new ArrayCollection(genData());
     }

     public function genData():Array {
        var result:Array = [];

        for (var i:int=0;i<dsLength;i++) {
           var localVals:Object = {
                valueA:Math.random()*100,
                valueB:Math.random()*100,
                valueX:Math.random()*100,
                valueY:Math.random()*100
           };

           // Push new object onto the data array.
           result.push(localVals);
        }
        return result;
     }
  ]]></mx:Script>

  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart" dataProvider="{dataSet}">
        <mx:series>
           <mx:PlotSeries
                xField="valueX"
                yField="valueY"
                displayName="Series 1"
           />
```

```
            <mx:PlotSeries
                 xField="valueA"
                 yField="valueB"
                 displayName="Series 2"
            />
        </mx:series>
    </mx:PlotChart>
    <mx:Legend id="l1" dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Changing chart data at run time

Using ActionScript, you can change a charting control's data at run time by using a variety of methods.

You can change a chart or a series data provider. The following example binds the data provider to a local variable. It then toggles the chart's data provider using that local variable when the user clicks the button.

```
<?xml version="1.0"?>
<!-- charts/ChangeDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);

     [Bindable]
     public var expenses2:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2400, Expenses:1509, Amount:950},
        {Month:"Feb", Profit:3000, Expenses:2200, Amount:400},
        {Month:"Mar", Profit:3500, Expenses:1200, Amount:200}
     ]);

     [Bindable]
     public var dp:ArrayCollection = expenses;

     public function changeDataProvider():void {
        if (dp==expenses) {
           dp = expenses2;
        } else {
           dp = expenses;
        }
     }
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="myChart" dataProvider="{dp}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{dp}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
```

```
                displayName="Expenses"
            />
            <mx:LineSeries
                yField="Amount"
                displayName="Amount"
            />
        </mx:series>
    </mx:LineChart>
    <mx:Legend dataProvider="{myChart}"/>
    <mx:Button label="Change Data Provider"
        click="changeDataProvider()"
    />
  </mx:Panel>
</mx:Application>
```

You can add or remove a data point from a series. The following example adds an item to the existing data provider when the user clicks the button:

```
<?xml version="1.0"?>
<!-- charts/AddDataItem.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var dpac:ArrayCollection =  new ArrayCollection([
           {A:2000},
           {A:3000},
           {A:4000},
           {A:4000},
           {A:3000},
           {A:2000},
           {A:6000}
        ]);

        public function addDataItem():void {
            var o:Object = {"A":2000};
            dpac.addItem(o);
        }
    ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        height="400"
        width="800"
        dataProvider="{dpac}"
     >
        <mx:series>
            <mx:ColumnSeries yField="A" displayName="Series 1"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
     <mx:Button label="Add Data Item" click="addDataItem();"/>
  </mx:Panel>
</mx:Application>
```

You can also change the fields of a series to change chart data at run time. You do this by changing the value of the data provider field (such as `xField` or `yField`) on the series object. To get a reference to the series, you use the series's `id` property or the chart control's series index. The following example toggles the data series when the user clicks on the Change Series button:

```
<?xml version="1.0"?>
<!-- charts/ToggleSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"initApp();">
  <mx:Script><![CDATA[
     [Bindable]
     public var dataSet:Array;
     public var myStates:Array =
        ["Wisconsin","Ohio","Arizona","Penn"];
     public var curSeries:String;
     public function initApp():void {
        var newData:Array = [];
        for(var i:int=0;i<myStates.length;i++) {
           newData[i] = {
            Apples: Math.floor(Math.random()*150),
            Oranges: Math.floor(Math.random()*150),
            myState: myStates[i]
           }
        }
        dataSet = newData;
        curSeries = "apples";
     }
     public function changeData():void {
        var series:Object = myChart.series[0];
        if (curSeries == "apples") {
           curSeries="oranges";
           series.yField = "Oranges";
           series.displayName = "Oranges";
           series.setStyle("fill",0xFF9933);
        } else {
           curSeries="apples";
           series.yField = "Apples";
           series.displayName = "Apples";
           series.setStyle("fill",0xFF0000);
        }
     }
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{dataSet}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
```

```
            <mx:CategoryAxis
                 dataProvider="{dataSet}"
                 categoryField="myState"
            />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSeries
                 yField="Apples"
                 displayName="Apples"
             >
             <mx:fill>
                 <mx:SolidColor color="0xFF0000"/>
             </mx:fill>
            </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
     <mx:Button id="b1"
        label="Change Series"
        click="changeData()"
     />
  </mx:Panel>
</mx:Application>
```

You can take advantage of data binding to make your chart reflect data changes in real time. The following example uses a Timer to define the intervals at which it checks for new data. Because the chart's data provider is bound to an ArrayCollection, whenever a new data point is added to the collection, the chart is automatically updated.

```
<?xml version="1.0"?>
<!-- charts/WatchingCollections.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initData();">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var dataSet:ArrayCollection;

        [Bindable]
        public var revenue:Number = 100;

        private var t:Timer;

        private function initData():void {
            dataSet = new ArrayCollection();
            t = new Timer(500);
            t.addEventListener(TimerEvent.TIMER, addData);
            t.start();
        }

        private function addData(e:Event):void {
            dataSet.addItem( { revenue: revenue } );
            revenue += Math.random() * 10 - 5;
        }
    ]]></mx:Script>

    <mx:SeriesInterpolate id="interp"
        elementOffset="0"
        duration="300"
        minimumElementDuration="0"
    />

    <mx:Panel title="Line Chart">
        <mx:LineChart id="myChart" dataProvider="{dataSet}">
            <mx:series>
                <mx:LineSeries
                    yField="revenue"
                    showDataEffect="{interp}"
                />
            </mx:series>
            <mx:horizontalAxis>
                <mx:LinearAxis autoAdjust="false"/>
            </mx:horizontalAxis>
```

```
        </mx:LineChart>
    </mx:Panel>
</mx:Application>
```

You can also use system values to update charts at run time. The following example tracks the value of the `totalMemory` property in a LineChart control in real time:

```
<?xml version="1.0"?>
<!-- charts/MemoryGraph.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"initTimer()">
  <mx:Script><![CDATA[
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  import mx.collections.ArrayCollection;

  [Bindable]
  public var memoryUsage:ArrayCollection = new ArrayCollection();

        public function initTimer():void {
           // The first parameter in the Timer constructor
           // is the interval, in milliseconds. The second
           // parameter is how many times to run (0 is
           // infinity).
           var myTimer:Timer = new Timer(1000, 0);

           // Add the listener for the timer event.
           myTimer.addEventListener("timer", timerHandler);
           myTimer.start();
        }

        public function timerHandler(event:TimerEvent):void {
           var o:Object = new Object();

           // Get the number of milliseconds since Flash
           // Player started.
           o.time = getTimer();

           // Get the total memory Flash Player is using.
           o.memory = flash.system.System.totalMemory;

           // Add new object to the ArrayCollection, which
           // is bound to the chart's data provider.
           memoryUsage.addItem(o);
        }
  ]]></mx:Script>

  <mx:LineChart id="chart"
        dataProvider="{memoryUsage}"
        showDataTips="true"
  >
     <mx:horizontalAxis>
        <mx:LinearAxis/>
     </mx:horizontalAxis>
```

```
      <mx:verticalAxis>
         <mx:LinearAxis minimum="5000000"/>
      </mx:verticalAxis>
      <mx:series>
         <mx:Array>
            <mx:LineSeries yField="memory"/>
         </mx:Array>
      </mx:series>
   </mx:LineChart>
</mx:Application>
```

CHAPTER 54

Chart Types

54

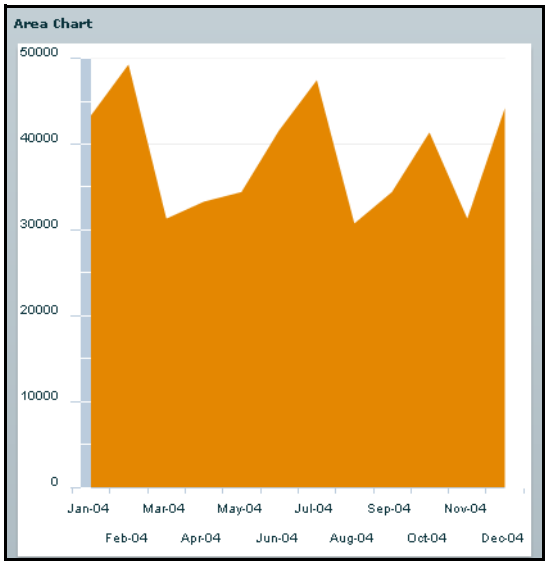
Adobe Flex provides different types of charting controls.

Contents

Using area charts

You use the AreaChart control to represent data as an area bounded by a line connecting the values in the data. The area underneath the line is filled in with a color or pattern. You can use an icon or symbol to represent each data point along the line, or you can show a simple line without icons.

The following image shows an example of an area chart:



The following example creates an AreaChart control:

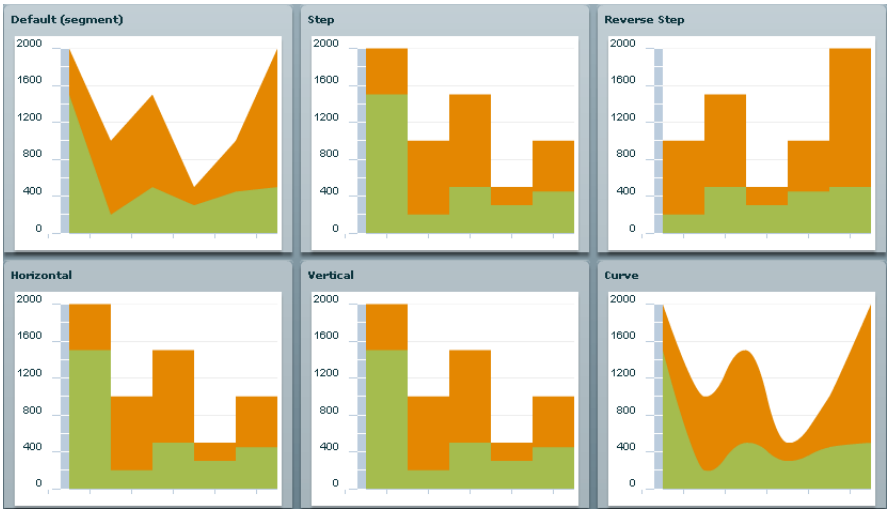
```
<?xml version="1.0"?>
<!-- charts/BasicArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

An area chart is essentially a line chart with the area underneath the line filled in; therefore, area charts and line charts share many of the same characteristics. For more information, see "Using line charts" on page 1646.

You use the AreaSeries chart series with the AreaChart control to define the data for the chart. The following table describes properties of the AreaSeries chart series that you commonly use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point. If you omit this field, Flex arranges the data points in the order of the data in the data provider.
`minField`	Specifies the field of the data provider that determines the y-axis location of the bottom of an area. This property is optional. If you omit it, the bottom of the area is aligned with the x-axis. This property has no effect on overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see "Using the minField property" on page 1726.
`form`	Specifies the way in which the data series is shown in the chart. The following values are valid: • `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default. • `step` Draws lines as horizontal and vertical segments. At the first data point, draws a horizontal line, and then a vertical line to the second point. Repeats this action for each data point. • `reverseStep` Draws lines as horizontal and vertical segments. At the first data point, draws a vertical line, and then a horizontal line to the second point. Repeats this action for each data point. • `vertical` Draws only the vertical line from the *y*-coordinate of the first point to *y*-coordinate of the second point at the *x*-coordinate of the second point. Repeats this action for each data point. • `horizontal` Draws only the horizontal line from the *x*-coordinate of the first point to x-coordinate of the second point at the *y*-coordinate of the first point. Repeats this action for each data point. • `curve` Draws curves between data points.

The following example shows the available forms for an AreaSeries series:



You can use the `type` property of the AreaChart control to represent a number of chart variations, including overlaid, stacked, 100% stacked, and high-low areas. For more information, see "Stacking charts" on page 1793.

Using bar charts

You use the BarChart control to represent data as a series of horizontal bars whose length is determined by values in the data. You can use the BarChart control to represent a number of chart variations, including clustered bars, overlaid, stacked, 100% stacked, and high-low areas. For more information, see "Stacking charts" on page 1793.

You use the BarSeries chart series with the BarChart control to define the data for the chart. The following table describes the properties of the BarSeries chart series that you use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of the base of each bar in the chart. If you omit this property, Flex arranges the bars in the order of the data in the data provider.

Property	Description
`xField`	Specifies the field of the data provider that determines the x-axis location of the end of each bar.
`minField`	Specifies the field of the data provider that determines the x-axis location of the base of a bar. This property has no effect in overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see "Using the minField property" on page 1726.

The following example creates a simple BarChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicBar.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

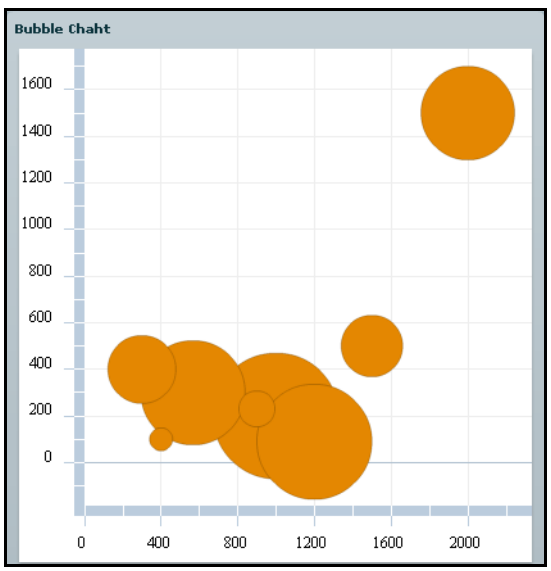
A bar chart is essentially a column chart rotated 90 degrees clockwise; therefore, bar charts and column charts share many of the same characteristics. For more information, see "Using column charts" on page 1635.

Using bubble charts

You use the BubbleChart control to represent data with three values for each data point: a value that determines its position along the x-axis, a value that determines its position along the y-axis, and a value that determines the size of the chart symbol, relative to the other data points on the chart.

The `<mx:BubbleChart>` tag takes an additional property, `maxRadius`. This property specifies the maximum radius of the largest chart element, in pixels. The data point with the largest value is assigned this radius; all other data points are assigned a smaller radius based on their value relative to the largest value. The default value is 30 pixels.

The following example shows a bubble chart:



You use the BubbleSeries chart series with the BubbleChart control to define the data for the chart. The following table describes the properties of the BubbleSeries chart series that you commonly use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point. This property is required.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point. This property is required.
`radiusField`	Specifies the field of the data provider that determines the radius of each symbol, relative to the other data points in the chart. This property is required.

The following example draws a BubbleChart control and sets the maximum radius of bubble elements to 50:

```
<?xml version="1.0"?>
<!-- charts/BasicBubble.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:120, Amount:45},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:60},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:30}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bubble Chart">
     <mx:BubbleChart id="myChart"
        maxRadius="50"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:BubbleSeries
                xField="Profit"
                yField="Expenses"
                radiusField="Amount"
                displayName="Profit"
           />
        </mx:series>
     </mx:BubbleChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

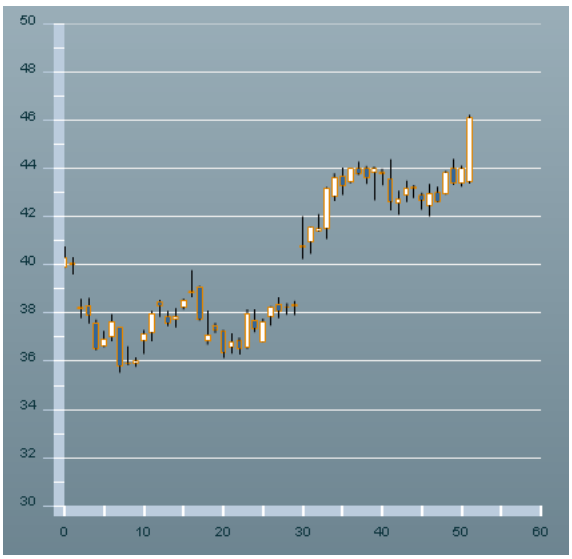
Using candlestick charts

The CandlestickChart control represents financial data as a series of candlesticks representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line in each candlestick represent the high and low values for the data point, while the top and bottom of the filled box represents the opening and closing values. Each candlestick is filled differently depending on whether the closing value for the data point is higher or lower than the opening value.

The CandlestickChart control's CandlestickSeries requires all four data points: high, low, open, and close. If you do not want to use opening value data points, you can use the HighLowOpenClose charts, which do not require a data point that represents the opening value. For more information, see "Using HighLowOpenClose charts" on page 1642.

You use the CandlestickSeries chart series with the CandlestickChart control to define the data.

The following shows an example of a CandlestickChart chart:

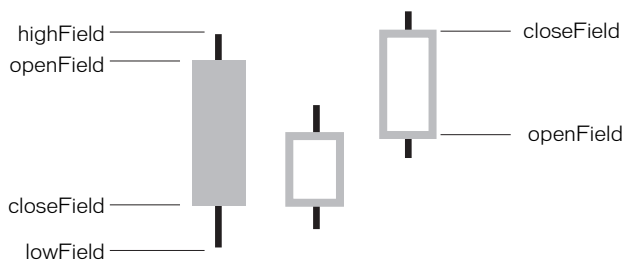


The following table describes the properties of the CandlestickSeries chart series that you commonly use to define your chart:

Property	Description
`closeField`	Specifies the field of the data provider that determines the y-axis location of the closing value of the element. This property defines the top or bottom of the candlestick.
`highField`	Specifies the field of the data provider that determines the y-axis location of the high value of the element. This property defines the top of the line inside the candlestick.
`lowField`	Specifies the field of the data provider that determines the y-axis location of the low value of the element. This property defines the bottom of the line inside the candlestick.
`openField`	Specifies the field of the data provider that determines the y-axis location of the opening value of the element. This property defines the position of the top or bottom of the candlestick.
`xField`	Specifies the field of the data provider that determines the x-axis location of the element. If set to the empty string (""), Flex renders the columns in the order in which they appear in the data provider. The default value is the empty string.

If the `closeField` is lower than the `openField`, Flex applies a solid fill to the candle. The color of this solid fill defaults to the color of the box's outline. It is defined by the `declineFill` style property. If the `closeField` is *higher* than the `openField`, Flex fills the candle with white by default.

The following image shows these properties. As you can see, the location of the `closeField` property can be either the top or the bottom of the candlestick, depending on whether it is higher or lower than the `openField` property:



The following example creates a CandlestickChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicCandlestick.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     [Bindable]
     public var TICKER:Array = [
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1}
     ];
  ]]></mx:Script>

  <mx:Panel title="Candlestick Chart">
     <mx:CandlestickChart id="mychart"
        dataProvider="{TICKER}"
        showDataTips="true"
        height="400"
        width="400"
     >
        <mx:series>
           <mx:CandlestickSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
           />
        </mx:series>
     </mx:CandlestickChart>
     <mx:Legend dataProvider="{mychart}"/>
  </mx:Panel>
</mx:Application>
```

You can change the color of the candle's fill with the `fill` and `declineFill` properties of the series. The `fill` property defines the color of the candlestick when the `closeField` value is higher than the `openField` value. The `declineFill` property defines the color of the candlestick when the reverse is true. You can also define the properties of the high-low lines and candlestick borders by using a Stroke class, as the following example shows:

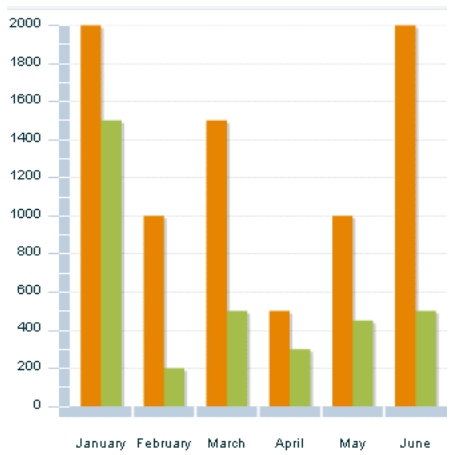
```
<?xml version="1.0"?>
<!-- charts/CandlestickStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var TICKER:ArrayCollection = new ArrayCollection([
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Candlestick Chart">
     <mx:CandlestickChart id="mychart"
        dataProvider="{TICKER}"
        showDataTips="true"
        height="400"
        width="400"
     >
        <mx:verticalAxis>
           <mx:LinearAxis title="linear axis" minimum="40" maximum=
           "50"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:CandlestickSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
           >
            <mx:fill>
```

```
                <mx:SolidColor color="green"/>
            </mx:fill>
            <mx:declineFill>
                <mx:SolidColor color="red"/>
            </mx:declineFill>
            <mx:stroke>
                <mx:Stroke weight="1" color="black"/>
            </mx:stroke>
          </mx:CandlestickSeries>
        </mx:series>
    </mx:CandlestickChart>
    <mx:Legend dataProvider="{mychart}"/>
  </mx:Panel>
</mx:Application>
```

Using column charts

The ColumnChart control represents data as a series of vertical columns whose height is determined by values in the data. You can use the ColumnChart control to create several variations of column charts, including simple columns, clustered columns, overlaid, stacked, 100% stacked, and high-low. For more information, see "Stacking charts" on page 1793.

The following example shows a ColumnChart control with two series:



You use the ColumnSeries chart series with the ColumnChart control to define the data for the chart. The following table describes the properties of the ColumnSeries chart series to define your chart:

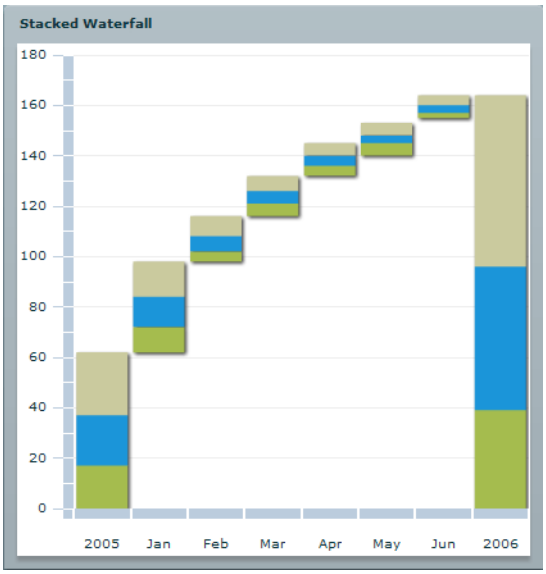
Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of the top of a column. This field defines the height of the column.
`xField`	Specifies the field of the data provider that determines the x-axis location of the column. If you omit this property, Flex arranges the columns in the order of the data in the data provider.
`minField`	Specifies the field of the data provider that determines the y-axis location of the bottom of a column. This property has no effect on overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see "Using the minField property" on page 1726.

The following example creates a ColumnChart control with two series:

```
<?xml version="1.0"?>
<!-- charts/BasicColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can set the `type` property on a ColumnChart control to stack and group series data in the chart. For more information, see "Stacking charts" on page 1793.

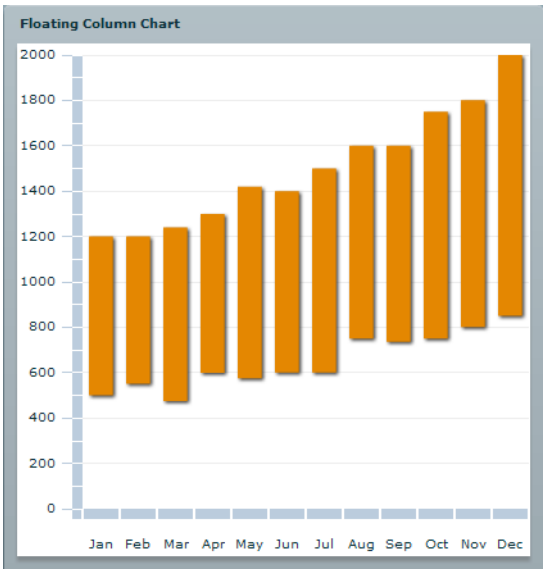
You can create cascading or waterfall column charts by using the ColumnChart control. One way to do this is to create an invisible series and use that to set the variable height of the other columns, creating the waterfall effect. The following is an example of a waterfall chart:



The following code creates this chart:

```
<?xml version="1.0"?>
<!-- charts/WaterfallStacked.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"2005", top:25, middle:20, bottom:17, Invisible:0},
        {Month:"Jan", top:14, middle:12, bottom:10, Invisible:62},
        {Month:"Feb", top:8, middle:6, bottom:4, Invisible:98},
        {Month:"Mar", top:6, middle:5, bottom:5, Invisible:116},
        {Month:"Apr", top:5, middle:4, bottom:4, Invisible:132},
        {Month:"May", top:5, middle:3, bottom:5, Invisible:140},
        {Month:"Jun", top:4, middle:3, bottom:2, Invisible:155},
        {Month:"2006", top:68, middle:57, bottom:39, Invisible:0}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Stacked Waterfall">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        columnWidthRatio=".9"
        showDataTips="true"
        type="stacked"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Invisible"
                displayName="Invisible"
           >
                <mx:fill>
                    <!--Set alpha to 0 to hide invisible column.-->
                    <mx:SolidColor color="0xFFFFFF" alpha="0"/>
                </mx:fill>
           </mx:ColumnSeries>
           <mx:ColumnSeries yField="bottom" displayName="Profit"/>
           <mx:ColumnSeries yField="middle" displayName="Expenses"/>
           <mx:ColumnSeries yField="top" displayName="Profit"/>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

You can also create floating column charts by using the `minField` property of the chart's data series. This property lets you set the lower level of a column. The following example shows a floating ColumnChart control:



The following code draws this chart:

```
<?xml version="1.0"?>
<!-- charts/MinFieldColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Revenue:1200, Expenses:500},
        {Month:"Feb", Revenue:1200, Expenses:550},
        {Month:"Mar", Revenue:1240, Expenses:475},
        {Month:"Apr", Revenue:1300, Expenses:600},
        {Month:"May", Revenue:1420, Expenses:575},
        {Month:"Jun", Revenue:1400, Expenses:600},
        {Month:"Jul", Revenue:1500, Expenses:600},
        {Month:"Aug", Revenue:1600, Expenses:750},
        {Month:"Sep", Revenue:1600, Expenses:735},
        {Month:"Oct", Revenue:1750, Expenses:750},
        {Month:"Nov", Revenue:1800, Expenses:800},
        {Month:"Dec", Revenue:2000, Expenses:850}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Floating Column Chart">
     <mx:ColumnChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Revenue"
                minField="Expenses"
                displayName="Revenue"
            />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

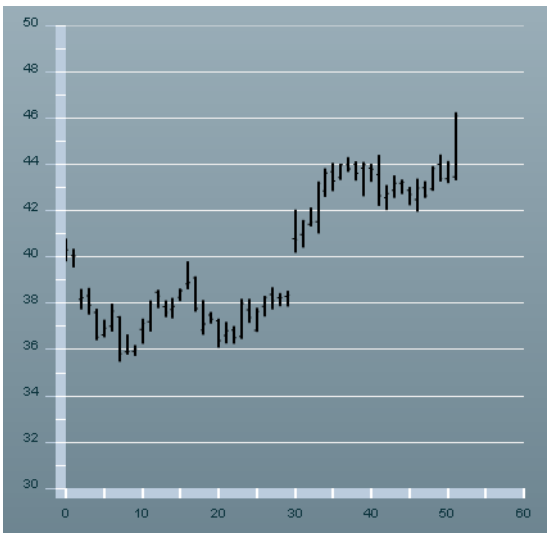
For more information, see "Using the minField property" on page 1726.

Using HighLowOpenClose charts

The HLOCChart control represents financial data as a series of lines representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line represent the high and low values for the data point, while the left tick mark represents the opening values and the right tick mark represents the closing values.

The HLOCChart control does not require a data point that represents the opening value. A related chart is the CandlestickChart control that represents similar data as candlesticks. For more information see "Using candlestick charts" on page 1631.

You use the HLOCSeries with the HLOCChart control to define the data for HighLowOpenClose charts. The following example shows an HLOC chart:

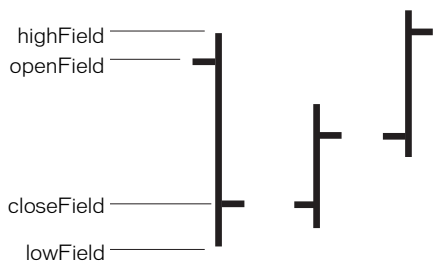


The following table describes the properties of the HLOCChart control's series that you commonly use to define your chart:

Property	Description
`closeField`	Specifies the field of the data provider that determines the *y*-axis location of the closing value of the element. This property defines the position of the right tick mark on the vertical line.
`highField`	Specifies the field of the data provider that determines the *y*-axis location of the high value of the element. This property defines the top of the vertical line.

Property	Description
`lowField`	Specifies the field of the data provider that determines the *y*-axis location of the low value of the element. This property defines the bottom of the vertical line.
`openField`	Specifies the field of the data provider that determines the y-axis location of the opening value of the element. This property defines the position of the left tick mark on the vertical line. This property is optional.
`xField`	Specifies the field of the data provider that determines the x-axis location of the element. If set to the empty string (""), Flex renders the columns in the order in which they appear in the data provider. The default value is the empty string.

Data points in an HLOCChart control do not require an `openField` property. If no `openField` property is specified, Flex renders the data point as a flat line with a single closing value indicator pointing to the right. If an `openField` property is specified, Flex renders the data point with another indicator pointing to the left, as the following image shows:



The following example creates an HLOCChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicHLOC.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var TICKER:ArrayCollection = new ArrayCollection([
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1}
     ]);
  ]]></mx:Script>

  <mx:Panel title="HighLowOpenClose Chart">
     <mx:HLOCChart id="myChart"
        dataProvider="{TICKER}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="50"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:HLOCSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
            >
           </mx:HLOCSeries>
        </mx:series>
     </mx:HLOCChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can change the stroke of the vertical lines by using a Stroke object on the series. To change the appearance of the opening and closing value tick marks on the vertical line, you use the `openTickStroke` and `closeTickStroke` style properties. The following example changes the stroke of the vertical line to 2 (the default value is 1) and the color of all the lines to black:

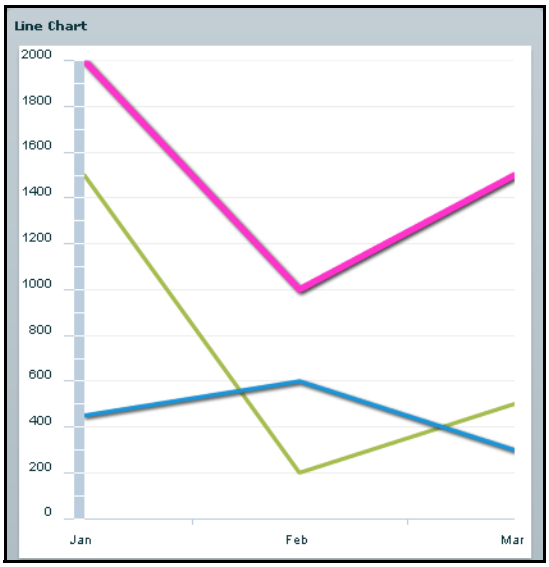
```
<?xml version="1.0"?>
<!-- charts/HLOCStyled.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var TICKER:ArrayCollection = new ArrayCollection([
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1},
     ]);
  ]]></mx:Script>

  <mx:Panel title="HLOC Chart">
     <mx:HLOCChart id="myChart"
        dataProvider="{TICKER}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="50"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:HLOCSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
            >
            <mx:stroke>
                <mx:Stroke color="#000000" weight="2"/>
            </mx:stroke>
            <mx:closeTickStroke>
                <mx:Stroke color="#000000" weight="1"/>
```

```
            </mx:closeTickStroke>
            <mx:openTickStroke>
                <mx:Stroke color="#000000" weight="1"/>
            </mx:openTickStroke>
          </mx:HLOCSeries>
        </mx:series>
      </mx:HLOCChart>
      </mx:Panel>
      <mx:Legend dataProvider="{myChart}"/>
</mx:Application>
```

Using line charts

The LineChart control represents data as a series of points, in Cartesian coordinates, connected by a continuous line. You can use an icon or symbol to represent each data point along the line, or show a simple line without icons.

The following example shows a simple line chart:

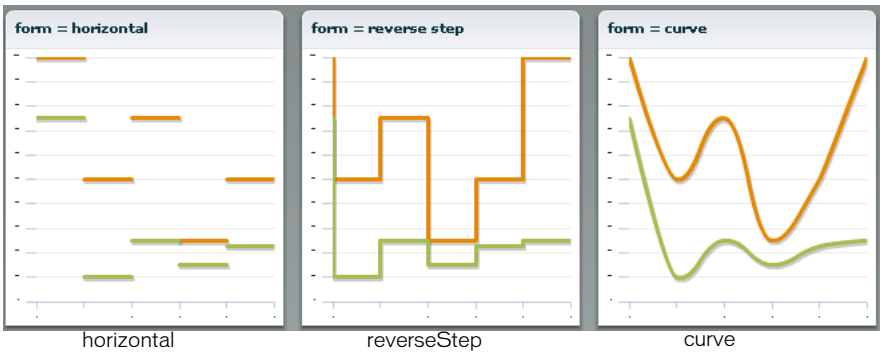


You use the LineSeries chart series with the LineChart control to define the data for the chart. The following table describes the properties of the LineSeries chart series you commonly use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point. This is the height of the line at that location along the axis.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point. If you omit this field, Flex arranges the data points in the order of the data in the data provider.
`interpolateValues`	Specifies how to represent missing data. If you set the value of this property to `false`, the chart breaks the line at the missing value. If you specify a value of `true`, Flex draws a continuous line by interpolating the missing value. The default value is `false`.
`form`	Specifies the way in which the data series is shown in the chart. The following values are valid: • `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default. • `step` Draws lines as horizontal and vertical segments. At the first data point, draws a horizontal line, and then a vertical line to the second point. Repeats this for each data point. • `reverseStep` Draws lines as horizontal and vertical segments. At the first data point, draws a vertical line, and then a horizontal line to the second point. Repeats this for each data point. • `vertical` Draws the vertical line only from the *y*-coordinate of the first point to the *y*-coordinate of the second point at the *x*-coordinate of the second point. Repeats this for each data point. • `horizontal` Draws the horizontal line only from the *x*-coordinate of the first point to the *x*-coordinate of the second point at the *y*-coordinate of the first point. Repeats this for each data point. • `curve` Draws curves between data points.

The following example shows the available forms for a LineSeries chart:

segment step vertical



horizontal reverseStep curve

The following example creates a LineChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicLine.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Formatting lines

You can change the width and color of the lines for each series by using the `<mx:lineStroke>` tag. The default line is 3 pixels wide and has a shadow. The following example sets a custom color and width for the series Stroke object:

```
<?xml version="1.0"?>
<!-- charts/BasicLineStroke.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart With Strokes">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
            yField="Profit"
            displayName="Profit"
           >
            <mx:lineStroke>
                <mx:Stroke
                    color="0x0099FF"
                    weight="20"
                    alpha=".2"
                />
            </mx:lineStroke>
           </mx:LineSeries>
           <mx:LineSeries
            yField="Expenses"
            displayName="Expenses"
           >
            <mx:lineStroke>
                <mx:Stroke
                    color="0x0044EB"
                    weight="20"
```

```
                    alpha=".8"
                />
            </mx:lineStroke>
          </mx:LineSeries>
        </mx:series>
      </mx:LineChart>
      <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

For more information on using the Stroke class in charts, see "Using strokes" on page 1728.

The default appearance of the lines in a LineChart control is with drop shadows. You can remove these shadows by setting the chart control's `seriesFilters` property to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LineChartNoShadows.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Line Chart with No Shadows">
     <mx:LineChart id="myChart" dataProvider="{expenses}">
        <mx:seriesFilters>
           <mx:Array/>
        </mx:seriesFilters>
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
           <mx:LineSeries
                yField="Amount"
                displayName="Amount"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also set the value of the `seriesFilters` property programmatically, as the following example shows:

```
myLineChart.seriesFilters = [];
```

You can also specify a programmatic renderer (or *skin*) class for each series by setting the `lineSegmentRenderer` property of the LineSeries. The default renderer is the LineRenderer, but Flex also applies a shadow filter on all series. If you remove the shadow filter, as the previous example shows, but want a line with a drop shadow in your chart, you can set the `lineSegmentRenderer` to the ShadowLineRenderer class, as the following example shows:

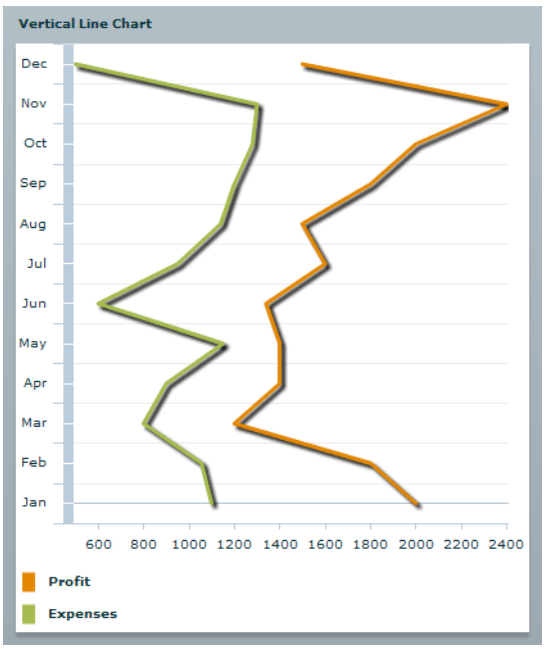
```
<?xml version="1.0"?>
<!-- charts/LineChartOneShadow.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Line Chart with One Shadow">
     <mx:LineChart id="myChart" dataProvider="{expenses}">
        <mx:seriesFilters>
           <mx:Array/>
        </mx:seriesFilters>
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
           <mx:LineSeries
                yField="Amount"
                displayName="Amount"
                lineSegmentRenderer=
                "mx.charts.renderers.ShadowLineRenderer"
            />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
```

```
</mx:Application>
```

For more information on using renderer classes to change the appearance of ChartItem objects such as the LineChart control's line segments, see "Skinning ChartItem objects" on page 1773.

Using vertical lines in a LineChart control

You can create LineChart controls that show vertical progression. The following example shows two LineSeries in a LineChart control that are displayed vertically rather than horizontally:



To make lines in a LineChart control display vertically rather than horizontally, you must do the following:

- Explicitly define the `xField` and `yField` properties for the LineSeries object.
- Set the `sortOnXField` property of the LineSeries object to `false`.

By default, data points in a series are sorted from left to right (on the x-axis) before rendering. This causes the LineSeries to draw horizontally. When you disable the `xField` sort and explicitly define a `yField` property, Flex draws the lines vertically rather than horizontally.

Flex does not sort any data vertically. As a result, you must ensure that your data is in order in the data provider. If it is not in order, Flex renders a zig-zagging line up and down the chart as it connects that dots according to position in the data provider.

The following example creates a LineChart control that displays vertical lines:

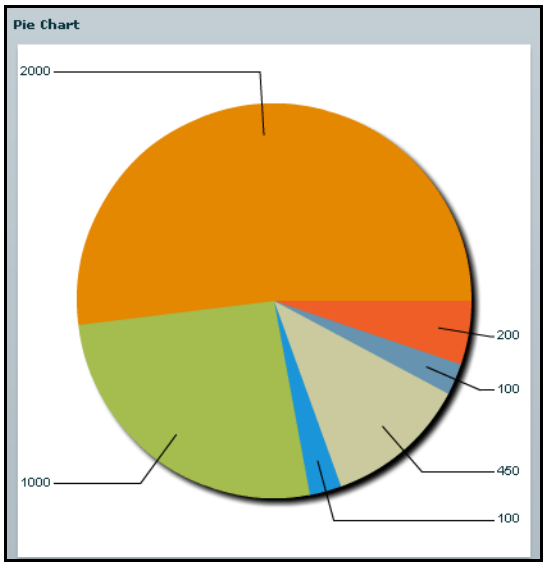
```
<?xml version="1.0"?>
<!-- charts/VerticalLineChart.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1100},
        {Month:"Feb", Profit:1800, Expenses:1055},
        {Month:"Mar", Profit:1200, Expenses:800},
        {Month:"Apr", Profit:1400, Expenses:900},
        {Month:"May", Profit:1400, Expenses:1150},
        {Month:"Jun", Profit:1340, Expenses:600},
        {Month:"Jul", Profit:1600, Expenses:950},
        {Month:"Aug", Profit:1500, Expenses:1140},
        {Month:"Sep", Profit:1800, Expenses:1200},
        {Month:"Oct", Profit:2000, Expenses:1280},
        {Month:"Nov", Profit:2400, Expenses:1300},
        {Month:"Dec", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Vertical Line Chart">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:LineSeries
                xField="Profit"
                yField="Month"
                displayName="Profit"
                sortOnXField="false"
           />
           <mx:LineSeries
                xField="Expenses"
                yField="Month"
                displayName="Expenses"
                sortOnXField="false"
           />
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
```

```
</mx:Application>
```

Using pie charts

You use the PieChart control to define a standard pie chart. The data for the data provider determines the size of each wedge in the pie chart relative to the other wedges.

The following example shows a pie chart:



You use the PieSeries chart series with the PieChart control to define the data for the chart. The PieSeries can create standard pie charts or doughnut charts. PieChart controls also support labels that identify data points.

The following table describes the properties of the PieChart control's PieSeries chart series that you commonly use to define your chart:

Property	Description
`field`	Specifies the field of the data provider that determines the data for each wedge of the pie chart.
`labelPosition`	Specifies how to render labels for the wedges.
`nameField`	Specifies the field of the data provider to use as the name for the wedge in DataTips and legends.

The following example defines a PieChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicPie.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Using labels with PieChart controls

PieChart controls support labels that display information about each data point. All charts support DataTips, which display the value of a data point when the user moves the mouse over it. Labels are different from DataTips in that they are always visible and do not react to mouse movements. The PieChart control is the only chart that supports labels.

To add labels to your PieChart control, set the `labelPosition` property on the series to a valid value other than `none`. To remove labels from your pie chart, set the `labelPosition` property to `none`. The default value is `none`.

The following table describes the valid values of the `labelPosition` property:

Value	Description
`callout`	Draws labels in two vertical stacks on either side of the PieChart control. Shrinks the PieChart if necessary to make room for the labels. Draws key lines from each label to the associated wedge. Shrinks labels as necessary to fit the space provided.
`inside`	Draws labels inside the chart. Shrinks labels to ensure that they do not overlap each other. Any label that must be drawn too small, as defined by the `insideLabelSizeLimit` property, is hidden from view.
`insideWithCallout`	Draws labels inside the pie, but if labels are shrunk below a legible size, Flex converts them to callout labels. This is a common value to set `labelPosition` to when the actual size of your chart is flexible and users might resize it.
`none`	Does not draw labels. This is the default value.
`outside`	Draws labels around the boundary of the PieChart control.

The following table describes the properties of the PieChart control that you can use to manipulate the appearance of labels:

Property	Description
`calloutGap`	Defines how much space, in pixels, to insert between the edge of the pie and the labels when rendering callouts. The default value is 10 pixels.
`calloutStroke`	Defines the line style used to draw the lines to callouts. For more information on defining line data points, see "Using strokes" on page 1728.
`insideLabelSizeLimit`	Defines the size threshold, expressed in points, below which inside labels are considered illegible. Below this threshold, labels are either removed entirely or turned into callouts based on the setting of the series `labelPosition` property.

To modify the label text, you use the `labelFunction` property to specify a callback function. The function specified in `labelFunction` returns a string that Flex displays as the label over the pie wedge. The following is the required format of the callback function:

```
function function_name ( data, field, index, percentValue ): return_type { }
```

The following table describes the parameters of the `labelFunction` callback function:

Parameter	Description
`data`	A reference to the data point that this pie wedge represents; type Object.
`field`	The field name from the data provide; type String.
`index`	The number of the data point in the data provider; type Number.
`percentValue`	The size of the pie wedge relative to the pie. If the pie wedge is a quarter of the size of the pie, this value is 25; type Number.

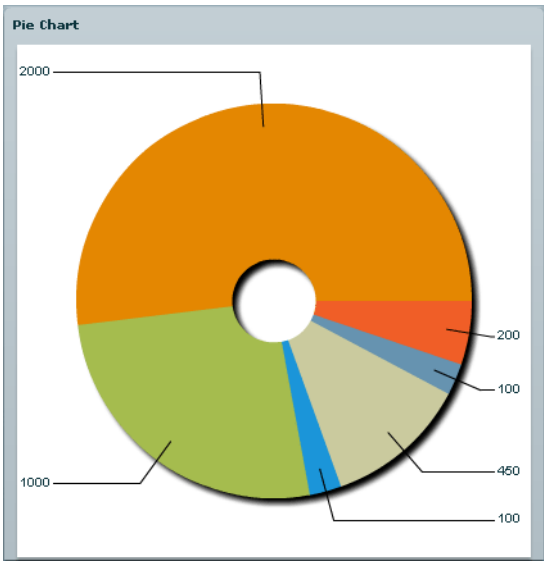
The following example generates labels that include data and formatting. It defines the `display()` method as the `labelFunction` to handle formatting of the label text.

```
<?xml version="1.0"?>
<!-- charts/PieLabelFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.formatters.*;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:1900000},
        {Expense:"Salaries", Amount:1350000},
        {Expense:"Building Rent", Amount:300000},
        {Expense:"Insurance", Amount:750000},
        {Expense:"Benefits", Amount:800000},
        {Expense:"Miscellaneous", Amount:900000}
     ]);
     public function display(
        data:Object,
        field:String,
        index:Number,
        percentValue:Number):String
     {
            return data.Expense + ":$" + data.Amount +
            "\n" + round(percentValue,2) + "%";
     }
     // Rounds to 2 places:
     public function round(num:Number, precision:Number):Number {
        var result:String;
        var f:NumberFormatter = new NumberFormatter();
        f.precision = precision;
        result = f.format(num);
        return Number(result);
     }
  ]]></mx:Script>
  <mx:Panel title="Expenditures for FY04">
     <mx:PieChart id="chart"
        dataProvider="{expenses}"
        showDataTips="false"
     >
        <mx:series>
           <mx:PieSeries
                labelPosition="callout"
                field="Amount"
                labelFunction="display"
           />
        </mx:series>
     </mx:PieChart>
  </mx:Panel>
```

```
</mx:Application>
```

Creating doughnut charts

Flex lets you create doughnut charts out of PieChart controls. Doughnut charts are identical to pie charts, except that they have hollow centers and resemble wheels rather than filled circles. The following example shows a doughnut chart:



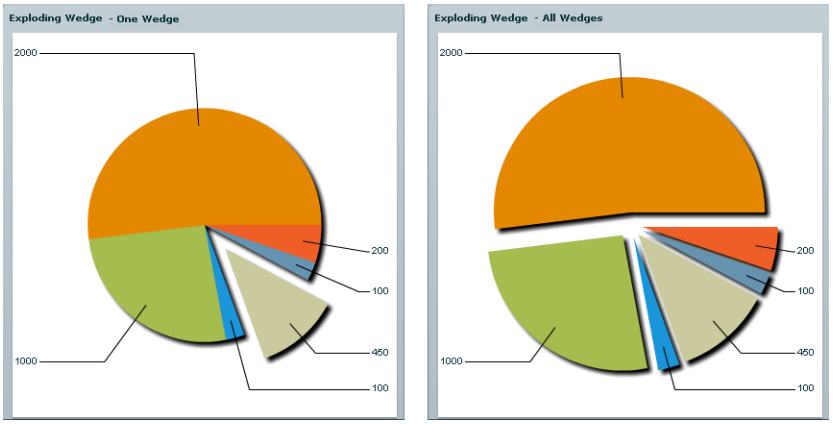
To create a doughnut chart, specify the `innerRadius` property on the PieChart control, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/DoughnutPie.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
        innerRadius=".3"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The value of the `innerRadius` property is a percentage value of the "hole" compared to the entire pie's radius. Valid values range from 0 to 1.

Creating exploding pie charts

The PieSeries chart series supports exploding wedges, both uniformly and on a per-wedge basis, so that you can achieve effects similar to the following:



The following table describes the properties that support exploding pie charts:

Property	Description
`explodeRadius`	A value from 0 to 1, representing the percentage of the available pie radius to use when exploding the wedges of the pie.
`perWedgeExplodeRadius`	An array of values from 0 to 1. The *N*th value in this array is added to the value of `explodeRadius` to determine the explode amount of each individual wedge of the pie. Individual values can be left undefined, in which case the wedge will only explode according to the `explodeRadius` property.
`reserveExplodeRadius`	A value from 0 to 1, representing an amount of the available pie radius to reserve for animating an exploding wedge.

To explode all wedges of a pie chart evenly, you use the `explodeRadius` property on the PieSeries, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ExplodingPie.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Exploding Pie Chart">
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <!--explodeRadius is a number between 0 and 1.-->
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                explodeRadius=".12"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

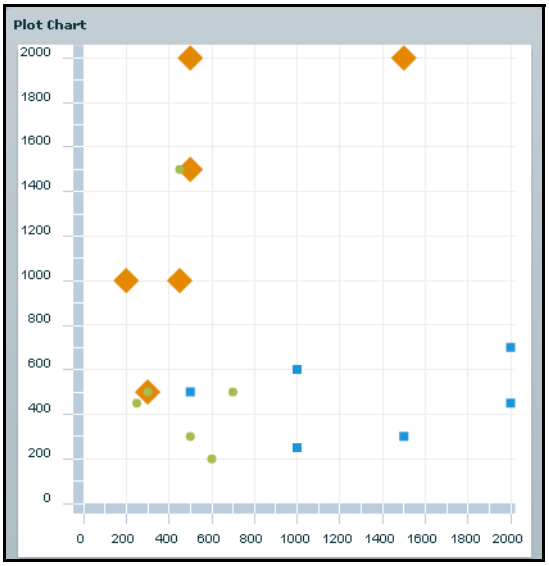
To explode one or more wedges of the pie, you use an Array of `explodeRadius` values. Each value in the Array applies to the corresponding data point. In the following example, the fourth data point, the Car expense, is exploded:

```
<?xml version="1.0"?>
<!-- charts/ExplodingPiePerWedge.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);

     // Create a bindable Array of explode radii.
     [Bindable]
     public var explodingArray:Array = [0,0,0,.2,0,0]
  ]]></mx:Script>
  <mx:Panel title="Exploding Pie Chart Per Wedge">
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <!--Apply the Array of radii to the PieSeries.-->
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                perWedgeExplodeRadius="{explodingArray}"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

Using plot charts

You use the PlotChart control to represent data in Cartesian coordinates where each data point has one value that determines its position along the x-axis, and one value that determines its position along the y-axis. You can define the shape that Flex displays at each data point with the renderer for the data series.

The following image shows an example of a plot chart:



You use the PlotSeries class with the PlotChart control to define the data for the chart. The following table describes the properties of the PlotSeries chart series that you commonly use to define your chart:

Property	Description
`yField`	Specifies the field of the data provider that determines the y-axis location of each data point.
`xField`	Specifies the field of the data provider that determines the x-axis location of each data point.
`radius`	Specifies the radius, in pixels, of the symbol at each data point. The default value is `5` pixels.

NOTE Both the `xField` and `yField` properties are required for each PlotSeries in a PlotChart control.

The following example defines three data series in a PlotChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicPlot.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
           />
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

By default, Flex displays the first data series in the chart as a diamond at each point. When you define multiple data series in a chart, Flex rotates the shape for the series (starting with a diamond, then a circle, then a square). If you have more series than there are default renderers, Flex begins again with the diamond.

The diamond shape, like the other shapes, is defined by a renderer class. The renderer classes that define these shapes are in the mx.charts.renderers package. The circle is defined by the CircleItemRenderer class. The following default renderer classes define the appearance of the data points:

- BoxItemRenderer
- CircleItemRenderer
- CrossItemRenderer
- DiamondItemRenderer
- ShadowBoxItemRenderer
- TriangleItemRenderer

You can control the image that is displayed by the chart for each data point by setting the `itemRenderer` style property of the series. The following example overrides the default renderers for the series:

```
<?xml version="1.0"?>
<!-- charts/PlotWithCustomRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
    ]]>
  </mx:Script>

  <mx:Panel title="Plot Chart With Custom Item Renderer">
     <mx:PlotChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"
                radius="10"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.DiamondItemRenderer"
                radius="10"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
                itemRenderer=
                "mx.charts.renderers.TriangleItemRenderer"
                radius="10"
           />
        </mx:series>
```

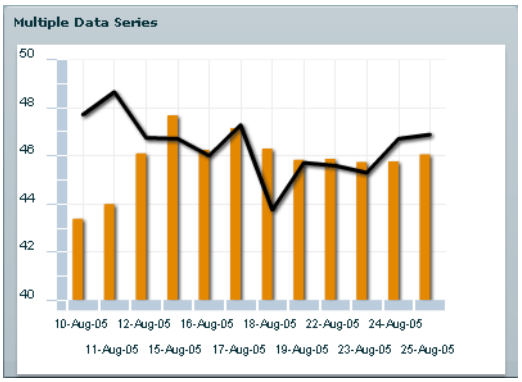
```
    </mx:PlotChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use graphics or custom classes to define each plot point. For more information, see "Creating custom renderers" on page 1775.

Using multiple data series

Charts that are subclasses of the CartesianChart class let you mix different data series in the same chart control. You can create a column chart with a trend line running through it or mix any data series with any other similar series.

The following chart tracks two stocks, one represented by a column and the other represented by a line:



You can use any combination of the following series objects in a CartesianChart control:

- AreaSeries
- BarSeries
- BubbleSeries
- CandlestickSeries
- ColumnSeries
- HLOCSeries
- LineSeries
- PlotSeries

The following example mixes a LineSeries and a ColumnSeries:

```
<?xml version="1.0"?>
<!-- charts/MultipleSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500"
height="600">
  <mx:Script>
    <![CDATA[
        [Bindable]
        public var SMITH:Array = [
           {date:"22-Aug-05", close:45.87},
           {date:"23-Aug-05", close:45.74},
           {date:"24-Aug-05", close:45.77},
           {date:"25-Aug-05", close:46.06},
        ];
        [Bindable]
        public var DECKER:Array = [
           {date:"22-Aug-05", close:45.59},
           {date:"23-Aug-05", close:45.3},
           {date:"24-Aug-05", close:46.71},
           {date:"25-Aug-05", close:46.88},
        ];
    ]]>
  </mx:Script>

  <mx:Panel title="Multiple Data Series" width="400" height="400">
     <mx:ColumnChart id="mychart"
        dataProvider="{SMITH}"
        showDataTips="true"
        height="250"
        width="350"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="date"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
           <mx:LinearAxis minimum="40" maximum="50"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:ColumnSeries
                dataProvider="{SMITH}"
                xField="date"
                yField="close"
                displayName="SMITH"
           >
           </mx:ColumnSeries>
           <mx:LineSeries
                dataProvider="{DECKER}"
                xField="date"
                yField="close"
                displayName="DECKER"
```

```
            >
            </mx:LineSeries>
        </mx:series>
    </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

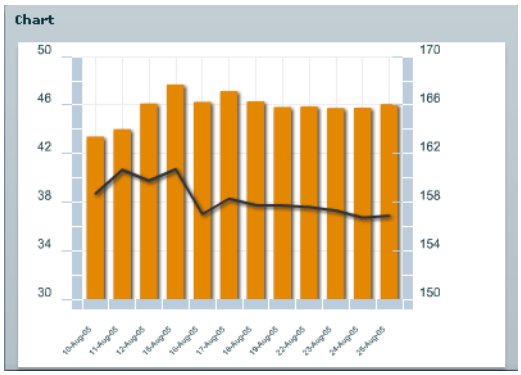
Using multiple series in the same chart works best when the data points are in a similar range (such as a stock price and its moving average). When the data points are in numerically very different ranges, the chart can be difficult to understand because the data is shown on a single axis. The solution to this problem is to use multiple axes, each with its own range. You can plot each data series on its own axis within the same chart using the techniques described in "Using multiple axes" on page 1673.

Using multiple axes

One potential problem when using two data series in a single chart is that if the scales of the data are very different, the data points might be plotted in very different areas on the chart's canvas. For example, one stock price could trade in the range of $100 to $150, while another stock price could fluctuate from $2 to $2.50. If you plot both stocks in the same chart, it would be difficult to see any correlation between the prices, even with a logarithmic axis.

To get around this problem, you use multiple axes in your charts so that each data series is positioned relative to its own axis. All chart controls that are subclasses of CartesianChart support adding a secondary set of data on a secondary scale in the horizontal axis, vertical axis, or both. (This applies to all charts except the PieChart control.) You can use values on a secondary axis to compare two sets of data that are on two different scales, such as stock prices that trade in different ranges.

The following example shows a stock price that trades within a $40 to 45 range, and another stock price that trades within a $150 to 160 range. The values of the axis on the left show the range of values of the first stock, and the values of the axis on the right show the range of values of the second stock:



By default, all charts that are subclasses of CartesianChart display a single set of data, and single horizontal and vertical axes. Setting any of the secondary properties, such as `secondDataProvider`, `secondSeries`, `secondVerticalAxis`/`secondHorizontalAxis`, or `secondVerticalAxisRenderer`/`secondHorizontalAxisRenderer`, has the following effect on the chart:

ColumnChart, LineChart, and AreaChart controls Primary and secondary series share the `horizontalAxis` by default, but each get a separate LinearAxis for their `verticalAxis`. A second `verticalAxisRenderer` is created by default when using multiple axes.

BarChart controls Primary and secondary series share the `verticalAxis`, but by default, each has a separate `horizontalAxis`. Two `horizontalAxisRenderers` are created by default.

PlotChart and BubbleChart controls Primary and secondary series have separate horizontal and vertical axes, and all four axis renderers. Bubble charts continue to share a single `radialAxis` among all series.

CartesianChart controls Cartesian charts have no extra axes or axis renderers by default.

PieChart controls Secondary series are not supported.

Use the `<mx:secondSeries>` tag to specify an Array of series objects that are rendered by using other scales. Any fields applied to series in this array refer to fields on the `secondDataProvider`.

You use the `secondDataProvider` property for the data provider used by any series provided in the `secondSeries` property.

The following example uses two series to allow a visual comparison of two stocks that trade in different ranges. It uses a single `categoryField` on the `horizontalAxis` for the horizontal values; it establishes minimum and maximum ranges of the tick marks for the `verticalAxis` and the `secondVerticalAxis`; and it uses the `secondVerticalAxisRenderer` to change the appearance of the secondary vertical axis.

```
<?xml version="1.0"?>
<!-- charts/MultipleAxes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
      public var SMITH:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:41.87},
        {date:"23-Aug-05", close:45.74},
        {date:"24-Aug-05", close:42.77},
        {date:"25-Aug-05", close:48.06},
     ]);

     [Bindable]
      public var DECKER:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:157.59},
        {date:"23-Aug-05", close:160.3},
        {date:"24-Aug-05", close:150.71},
        {date:"25-Aug-05", close:156.88},
     ]);

  ]]></mx:Script>

  <mx:Panel title="Column Chart With Multiple Series">
     <mx:ColumnChart id="myChart"
        dataProvider="{SMITH}"
        secondDataProvider="{DECKER}"
        showDataTips="true"
     >

        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{SMITH}"
                categoryField="date"
           />
        </mx:horizontalAxis>

        <mx:verticalAxis>
           <mx:LinearAxis minimum="40" maximum="50"/>
        </mx:verticalAxis>

        <mx:series>
```

```
            <mx:ColumnSeries id="cs1"
                 dataProvider="{SMITH}"
                 xField="date"
                 yField="close"
                 displayName="SMITH"
            />
         </mx:series>

         <mx:secondVerticalAxis>
            <mx:LinearAxis minimum="150" maximum="170"/>
         </mx:secondVerticalAxis>

         <mx:secondSeries>
            <mx:LineSeries id="cs2"
                 dataProvider="{DECKER}"
                 xField="date"
                 yField="close"
                 displayName="DECKER"
            />
         </mx:secondSeries>
      </mx:ColumnChart>
   </mx:Panel>
</mx:Application>
```

This example does not include a legend. When you create a chart with multiple axes, you cannot use the standard `<mx:Legend>` tag to create a legend for that chart. You must create a custom legend. For more information, see "Creating a custom Legend control" on page 1785.

Using the second axis

Depending on the chart type, secondary series may either share the same axes with the primary series or render against separate default secondary axes. Charts with horizontal and vertical axes use the `secondHorizontalAxis` and `secondVerticalAxis` properties to support the second series. If you explicitly provide a secondary horizontal or vertical axis, it is used by the secondary series, regardless of the chart type.

The following example defines the `secondVerticalAxis` as a LogAxis chart axis:

```
<mx:secondVerticalAxis>
   <mx:LogAxis/>
</mx:secondVerticalAxis>
```

As with primary axes, you can change the appearance of the secondary axes using renderers. In this case, you use the `<mx:secondVerticalAxisRenderer>` and `<mx:secondHorizontalAxisRenderer>` tags. The following example sets the location of the tick marks for the secondary axis:

```
<mx:secondVerticalAxisRenderer>
  <mx:AxisRenderer placement="left" tickPlacement="inside"/>
</mx:secondVerticalAxisRenderer>
```

For more information on formatting axes, see "Working with axes" on page 1698

Using multiple axis renderers

If a chart displays data with different vertical or horizontal scales, it must also render the multiple axes. You can specify where a particular axis renderer appears by setting the `placement` property of the axis renderer. Valid values are `left`, `top`, `right`, or `bottom`. The default value of the `placement` property of the primary renderer is `left` for vertical renderers and `bottom` for horizontal renderers. For secondary renderers, the default values are `right` and `top`.

You use the `secondHorizontalAxisRenderer` and `secondVerticalAxisRenderer` properties to control the appearance of the secondary axis, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/MultipleAxesMultipleRenderers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Script><![CDATA[
  [Bindable]
   public var SMITH:Array = [
     {date: "22-Aug-05", close: 45.87},
     {date: "23-Aug-05", close: 45.74},
     {date: "24-Aug-05", close: 45.77},
     {date: "25-Aug-05", close: 46.06},
  ];
  [Bindable]
  public var DECKER:Array = [
     {date: "22-Aug-05", close: 157.59},
     {date: "23-Aug-05", close: 157.3},
     {date: "24-Aug-05", close: 156.71},
     {date: "25-Aug-05", close: 156.88},
  ];
  ]]></mx:Script>
  <mx:Panel title="Chart" width="400" height="400">
     <mx:ColumnChart id="mychart"
        dataProvider="{SMITH}"
        secondDataProvider="{DECKER}"
        showDataTips="true"
        height="250"
        width="350"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{SMITH}"
                categoryField="date"
           />
        </mx:horizontalAxis>

        <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="50"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
                dataProvider="{SMITH}"
                xField="date"
                yField="close"
                displayName="SMITH"
           />
        </mx:series>
```

```
        <mx:secondVerticalAxis>
           <mx:LinearAxis minimum="150" maximum="170"/>
        </mx:secondVerticalAxis>

        <mx:verticalAxisRenderer>
           <mx:AxisRenderer
            placement="right"
            tickPlacement="inside"
           >
            <mx:axisStroke>
                <mx:Stroke color="#CC6699" weight="1"/>
            </mx:axisStroke>
            <mx:tickStroke>
                <mx:Stroke color="#CC0099" weight="1"/>
            </mx:tickStroke>
            <mx:minorTickStroke>
                <mx:Stroke color="#990066" weight="1"/>
            </mx:minorTickStroke>
           </mx:AxisRenderer>
        </mx:verticalAxisRenderer>

        <mx:secondVerticalAxisRenderer>
           <mx:AxisRenderer
            placement="left"
            tickPlacement="inside"
           >
            <mx:axisStroke>
                <mx:Stroke color="#000080" weight="1"/>
            </mx:axisStroke>
            <mx:tickStroke>
                <mx:Stroke color="#000160" weight="1"/>
            </mx:tickStroke>
            <mx:minorTickStroke>
                <mx:Stroke color="#100040" weight="1"/>
            </mx:minorTickStroke>
           </mx:AxisRenderer>
        </mx:secondVerticalAxisRenderer>

        <mx:secondSeries>
           <mx:LineSeries
                dataProvider="{DECKER}"
                xField="date"
                yField="close"
                displayName="DECKER"
           />
        </mx:secondSeries>

     </mx:ColumnChart>
  </mx:Panel>
```

```
</mx:Application>
```

Charts ensure that axis renderers do not overlay each other by automatically setting the `placement` property of the secondary axis renderers to lie opposite the primary axis renderers. If the `verticalAxisRenderer` sets the `placement` property to left, a `secondVerticalAxisRenderer` is placed on the right, and vice versa.

CHAPTER 55.

Formatting Charts

55

You can customize the look and feel of your Adobe Flex Charting components. You can format the appearance of almost all chart elements, from the font properties of an axis label to the width of the stroke in a legend.

To set style properties in your chart controls, you can use Cascading Style Sheets (CSS) syntax or set the style properties inline as tag attributes. As with all styles, you can call the `setStyle()` method to set any style on your chart elements. For more information on using the `setStyle()` method, see "Using the setStyle() and getStyle() methods" on page 737.

Contents

Applying chart styles

You can apply styles to charts by using CSS or inline syntax. You can also apply styles to chart elements by using binding.

Applying styles with CSS

You can apply styles to Flex Charting components with CSS definitions. You can set chart properties such as fonts and tick marks, or series properties, such as the fills of the boxes in a ColumnChart.

A limitation of using CSS to style your charts is that the styleable chart properties often use compound values, such as strokes and gradient fills, that cannot be expressed by using CSS. The result is that you cannot express all values of chart styles by using CSS syntax.

Applying CSS to chart controls

You can use the control name in CSS to define styles for that control. This is referred to as a type selector, because the style you define is applied to all controls of that type. For example, the following style definition specifies the font for all BubbleChart controls:

```
<?xml version="1.0"?>
<!-- charts/BubbleStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Style>
      BubbleChart {
         fontFamily:Arial;
         fontSize:20;
         color:#FF0033;
       }
    </mx:Style>
    <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:120, Amount:45},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:60},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:30}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bubble Chart">
     <mx:BubbleChart id="myChart"
        maxRadius="50"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
```

```
            <mx:BubbleSeries
                 xField="Profit"
                 yField="Expenses"
                 radiusField="Amount"
            />
        </mx:series>
    </mx:BubbleChart>
  </mx:Panel>
</mx:Application>
```

Some styles, such as `fontSize` and `fontFamily`, are inheritable, which means that if you set them on the chart control, the axes labels, titles, and other text elements on the chart inherit those settings. To determine whether a style is inheritable, see that style's description in *Adobe Flex 2 Language Reference*.

Axis labels appear next to the tick marks along the chart's axis. Titles appear parallel to the axis line. By default, the text on an axis uses the text styles of the chart control.

Axis elements use whatever styles are set on the chart control's type or class selector, but you can also specify different styles for each axis by using a class selector on the axis renderer or predefined axis style properties.

Applying different styles to each series

To apply different styles to each series in the charts with CSS, you use the `chartSeriesStyles` property. This property takes an Array of Strings. Each String specifies the name of a class selector in the style sheet. Flex applies each of these class selectors to a series.

To apply CSS to a series, you define a type or class selector for the chart that defines the `chartSeriesStyles` property. You then define each class selector named in the `chartSeriesStyles` property.

Essentially, you are defining a new style for each series in your chart. For example, if you have a ColumnChart control with two series, you can apply a different style to each series without having to explicitly set the styles on each series.

The following example defines the colors for two series in the ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/SeriesStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     ColumnChart {
        chartSeriesStyles: PCCSeries1, PCCSeries2;
     }
     .PCCSeries1 {
        fill: #CCFF66;
```

```
    }
    .PCCSeries2 {
       fill: #CCFF99;
    }
  </mx:Style>

  <mx:Script>
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month: "Jan", Profit: 2000, Expenses: 1500},
     {Month: "Feb", Profit: 1000, Expenses: 200},
     {Month: "Mar", Profit: 1500, Expenses: 500}
  ]);
  </mx:Script>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

Flex sets the `styleName` property of each series to the corresponding selector in the `chartSeriesStyles` Array. You do not have to explicitly set styles for each series. If you have more series than available `chartSeriesStyles` selectors, the chart begins again with the first style.

If you manually set the value of the `styleName` property on a particular series, that style takes priority over the styles that the `chartSeriesStyles` property specifies.

For PieChart controls, you can define the `fills` property of the series. The PieChart applies each of the values in this Array for each pie wedge. It starts with the first value if there are more wedges than values defined in the Array. The following example creates a PieChart that uses only red, white, and blue colors for the wedges:

```
<?xml version="1.0"?>
<!-- charts/PieWedgeFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
      PieSeries {
         fills:#FF0000, #FFFFFF, #006699;
      }
  </mx:Style>

  <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
           {Expense:"Taxes", Amount:2000},
           {Expense:"Rent", Amount:1000},
           {Expense:"Bills", Amount:100},
           {Expense:"Car", Amount:450},
           {Expense:"Gas", Amount:100},
           {Expense:"Food", Amount:200}
        ]);
    ]]>
  </mx:Script>
  <mx:Panel>
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

Using predefined axis style properties

The following predefined class selectors are available for axis styles:

- `horizontalAxisStyleName`
- `verticalAxisStyleName`
- `secondHorizontalAxisStyleName`
- `secondVerticalAxisStyleName`

Flex applies each style to the corresponding axis.

In addition to these class selectors for the axis style properties, you can use the `axisTitleStyleName` and `gridLinesStylesName` class selectors to apply styles to axis titles and grid lines.

The following example removes tick marks from the horizontal axis:

```
<?xml version="1.0"?>
<!-- charts/PredefinedAxisStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     ColumnChart {
        horizontalAxisStyleName:myAxisStyles;
        verticalAxisStyleName:myAxisStyles;
     }

     .myAxisStyles {
        tickPlacement:none;
     }
  </mx:Style>

  <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
           {Month:"Jan", Profit:2000, Expenses:1500},
           {Month:"Feb", Profit:1000, Expenses:200},
           {Month:"Mar", Profit:1500, Expenses:500}
        ]);
    ]]>
  </mx:Script>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
```

```
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
               displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

Using class selectors for axis styles

To use a class selector to define styles for axis elements, define the custom class selector in an `<mx:Style>` block or external style sheet, and then use the `styleName` property of the AxisRenderer class to point to that class selector.

The following example defines the `MyStyle` style and applies that style to the elements on the horizontal axis:

```
<?xml version="1.0"?>
<!-- charts/AxisClassSelectors.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

<mx:Style>
  .myStyle {
     fontSize:7;
     color:red;
  }
</mx:Style>

  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan",Profit:2000,Expenses:1500},
     {Month:"Feb",Profit:1000,Expenses:200},
     {Month:"Mar",Profit:1500,Expenses:500}
  ]);
  ]]></mx:Script>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
```

```
        <mx:horizontalAxisRenderer>
           <mx:AxisRenderer styleName="myStyle"/>
        </mx:horizontalAxisRenderer>

        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

Most charting-specific style properties are not inheritable, which means that if you set the property on a parent object, the child object does not inherit its value.

For more information on using CSS, see "About styles" on page 697.

Applying styles inline

You can set many styleable elements of a chart as attributes of the MXML tag. For example, to set the styleable properties of an axis, you can use the `<mx:AxisRenderer>` tag rather than define a new style in CSS.

The following example sets the `fontSize` property of the horizontal axis to 7:

```
<mx:horizontalAxisRenderer>
  <mx:AxisRenderer fontSize="7"/>
</mx:horizontalAxisRenderer>
```

You can also access the properties of renderers in ActionScript so that you can change their appearance at run time. For additional information about axis renderers, see "Working with axes" on page 1698.

Applying styles by binding tag definitions

You can define styles with MXML tags. You can then bind the values of the renderer properties to those tags. The following example defines the weight and color of the strokes, and then applies those strokes to the chart's AxisRenderer class:

```
<?xml version="1.0"?>
<!-- charts/BindStyleValues.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  [Bindable]
  public var aapl:Array = [
     {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:
     42.75},
     {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:
     43.19},
     {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:
     43.22},
     {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
     {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:
     42.99},
     {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
     {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:
     43.82},
     {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
     {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
     {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:
     46.1},
  ];
  ]]></mx:Script>

  <mx:Stroke color="0x00FF00" weight="2" id="axis"/>
  <mx:Stroke color="0xFF0000" weight="1" id="ticks"/>
  <mx:Stroke color="0x0000FF" weight="1" id="mticks"/>

  <mx:HLOCChart id="mychart"
    dataProvider="{aapl}"
    showDataTips="true"
  >
     <mx:horizontalAxisRenderer>
        <mx:AxisRenderer
            axisStroke="{axis}"
            placement="bottom"
            minorTickPlacement="inside"
            minorTickLength="2"
            tickLength="5"
            tickPlacement="inside"
        >
           <mx:tickStroke>{ticks}</mx:tickStroke>
           <mx:minorTickStroke>{mticks}</mx:minorTickStroke>
```

```
        </mx:AxisRenderer>
    </mx:horizontalAxisRenderer>

    <mx:verticalAxis>
        <mx:LinearAxis minimum="30" maximum="50"/>
    </mx:verticalAxis>
    <mx:series>
        <mx:HLOCSeries
            dataProvider="{aapl}"
            openField="open"
            highField="high"
            lowField="low"
            closeField="close"
            displayName="AAPL"
        >
        </mx:HLOCSeries>
    </mx:series>
  </mx:HLOCChart>
  <mx:Legend dataProvider="{mychart}"/>
</mx:Application>
```

Adding ChartElement objects

The ChartElement class is the base class for anything that appears in the data area of the chart. All series objects (such as GridLines objects) are ChartElement objects. You can add ChartElement objects (such as images, grid lines, and strokes) to your charts by using the `backgroundElements` and `annotationElements` properties of the chart classes.

The `backgroundElements` property specifies an Array of ChartElement objects that appear beneath any data series rendered by the chart. The `annotationElements` property specifies an Array of ChartElement objects that appears above any data series rendered by the chart.

The ChartElement objects that you can add to a chart include supported image files, such as GIF, SVG, and JPEG.

The following example adds new grid lines as annotation elements to the chart and an image as the background element. When the user clicks the button, the annotation elements change:

```
<?xml version="1.0"?>
<!-- charts/AnnotationElements.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  import mx.graphics.SolidColor;
  import mx.charts.GridLines;

  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
```

```
    {Month:"Jan", Profit:2000, Expenses:1500},
    {Month:"Feb", Profit:1000, Expenses:200},
    {Month:"Mar", Profit:1500, Expenses:500}
]);

[Embed(source="../assets/bird.gif")]
public var bird:Class;

public function updateGridLines():void {
    var bgi:GridLines = new GridLines();

    var s:Stroke = new Stroke(0xff00ff, 3);
    bgi.setStyle("horizontalStroke",s);

    var c:SolidColor = new SolidColor(0x990033, .2);
    bgi.setStyle("horizontalFill",c);

    var c2:SolidColor = new SolidColor(0x999933, .2);
    bgi.setStyle("horizontalAlternateFill",c2);

    myChart.annotationElements = [bgi]

    var b:Object = new bird();
    b.alpha = .2;
    b.height = 150;
    b.width = 150;

    myChart.backgroundElements = [ b ];
}
]]></mx:Script>
<mx:Panel>
    <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
            <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
```

```
        <mx:annotationElements>
           <mx:GridLines>
            <mx:horizontalStroke>
                <mx:Stroke
                    color="#191970"
                    weight="2"
                    alpha=".3"
                />
            </mx:horizontalStroke>
           </mx:GridLines>
        </mx:annotationElements>

        <mx:backgroundElements>
           <mx:Image
                source="@Embed('../assets/bird.gif')"
                alpha=".2"
           />
        </mx:backgroundElements>

     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
     <mx:Button id="b1"
        click="updateGridLines()"
        label="Update Grid Lines"
     />
  </mx:Panel>
</mx:Application>
```

In ActionScript, you can add an image and manipulate its properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BackgroundElementsWithActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="addBird()">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  import mx.graphics.SolidColor;
  import mx.charts.GridLines;

  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);

  [Embed(source="../assets/bird.gif")]
  public var bird:Class;
```

```
  public function addBird():void {
     var b:Object = new bird();
     b.alpha = .2;
     myChart.backgroundElements = [ b ];
  }
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

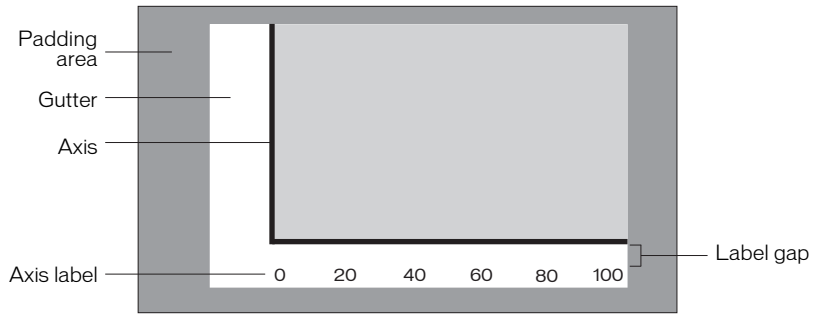
Setting padding properties

As with other Flex components, the padding properties of a chart define an area between the outside bounds of the chart control and its content. Flex draws only the background fill in the padding area.

You can set the padding values for a chart control by using the `paddingLeft` and `paddingRight` properties that the chart control inherits from the UIComponent class. You can also use the `paddingTop` and `paddingBottom` properties that the chart control inherits from the ChartBase class.

Flex Charting elements also have gutters. The *gutter* is the area between the padding area and the actual axis line. Flex draws labels, titles, and tick marks for the axes in the gutter of the chart. Chart controls adjust the gutters to accommodate these enhancements to the axis, but you can specify explicit gutter values.

The following example shows the locations of the gutters and the padding area on a chart:



The following style properties define the size of a chart's gutters:

- `gutterLeft`
- `gutterRight`
- `gutterTop`
- `gutterBottom`

The default value of the gutter styles is `undefined`, which means that the chart determines appropriate values. Overriding the default value and explicitly setting gutter values can improve the speed of rendering charts, because Flex does not have to dynamically calculate the gutter size. However, it can also cause clipping of axis labels or other undesirable effects.

You set gutter styles on the chart control. The following example creates a region that is 50 pixels wide for the axis labels, titles, and tick marks, by explicitly setting the values of the `gutterLeft`, `gutterRight`, and `gutterBottom` style properties. It also sets the `paddingTop` property to 20.

```
<?xml version="1.0"?>
<!-- charts/GutterStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     ColumnChart {
        gutterLeft:50;
        gutterRight:50;
        gutterBottom:50;
        paddingTop:20;
     }
  </mx:Style>

  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
```

```
    {Month:"Feb", Profit:1000, Expenses:200},
    {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>
  <mx:Panel>
    <mx:ColumnChart id="column" dataProvider="{expenses}">
      <mx:horizontalAxis>
        <mx:CategoryAxis
            dataProvider="{expenses}"
            categoryField="Month"
         />
      </mx:horizontalAxis>
      <mx:series>
        <mx:ColumnSeries
            xField="Month"
            yField="Profit"
            displayName="Profit"
        />
        <mx:ColumnSeries
            xField="Month"
            yField="Expenses"
            displayName="Expenses"
        />
      </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

Alternatively, you can set the gutter properties inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GutterProperties.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        gutterLeft="50"
        gutterRight="50"
        gutterBottom="50"
        gutterTop="20"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

In addition to the gutter and padding properties, you can set the `labelGap` property of a chart's axes. The `labelGap` property defines the distance, in pixels, between the tick marks and the axis labels. You set the `labelGap` property on the AxisRenderer tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LabelGaps.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
      import mx.collections.ArrayCollection;
      [Bindable]
      public var expenses:ArrayCollection = new ArrayCollection([
         {Month:"Jan", Profit:2000, Expenses:1500},
         {Month:"Feb", Profit:1000, Expenses:200},
         {Month:"Mar", Profit:1500, Expenses:500}
      ]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        gutterLeft="50"
        gutterRight="50"
        gutterBottom="50"
        gutterTop="20"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderer>
           <mx:AxisRenderer labelGap="20"/>
        </mx:horizontalAxisRenderer>
        <mx:verticalAxisRenderer>
           <mx:AxisRenderer labelGap="20"/>
        </mx:verticalAxisRenderer>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
```

```
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Working with axes

You can add axis titles and labels, format tick marks and the axis line, and rotate axis elements. Each chart, except for a pie chart, has horizontal and vertical axes. Along each axis, Flex draws small markers, called tick marks, with a corresponding label. That label can be a text string or a numeric value.

There are two axis types: category and numeric. A *category axis* typically defines strings that represent groupings of items in the chart; for example, types of expenses (such as rent, utilities, and insurance) or names of employees. A *numeric axis* typically defines continuous data such as the amount of an expense or the productivity gains of the employee. These data define the height of a column or width of a pie wedge, for example.

The two axis types share some common properties that you can use to define the appearance of the axis. For example, you can customize the values of the labels on the axis using the `labelFunction` property of the axis. This property is described in "Defining axis labels" on page 1715. You set the title of the axis using the `title` property as described in "Adding axis titles" on page 1711.

The following sections describe the CategoryAxis and NumericAxis classes.

About the CategoryAxis class

The CategoryAxis class maps discrete categorical data (such as states, product names, or department names) to an axis and spaces them evenly along it. This axis accepts any data type that can be represented by a String.

The `dataProvider` property of the CategoryAxis object defines the data provider that contains the text for the labels. In most cases, this can be the same data provider as the chart's data provider. A CategoryAxis object used in a chart inherits its `dataProvider` property from the containing chart, so you are not required to explicitly set the `dataProvider` property on a CategoryAxis object.

The `dataProvider` property of a CategoryAxis object can contain an Array of labels or an Array of objects. If the data provider contains objects, you use the `categoryField` property to point to the field in the data provider that contains the labels for the axis, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If the data provider contains an Array of labels only, you do not specify the `categoryField` property.

You can customize the labels of the CategoryAxis object rather than use the axis labels in the data provider. You do this by providing a custom data provider. To provide a custom data provider, set the value of the CategoryAxis object's `dataProvider` property to a custom Array of labels, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CategoryAxisLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);

     [Bindable]
     public var months:Array = [
        {Month:"January", monthAbbrev:"Jan"},
        {Month:"February", monthAbbrev:"Feb"},
        {Month:"March", monthAbbrev:"Mar"}
     ];


  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:AreaChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{months}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
  </mx:Panel>
</mx:Application>
```

For more information about chart data providers, see "Defining chart data" on page 1592.

You can also customize the labels using the `labelFunction` property of the CategoryAxis class to point to a function that refines the labels based on the existing data provider. For more information, see "Defining axis labels" on page 1715.

About the NumericAxis class

The NumericAxis class maps a set of continuous numerical values (such as sales volume, revenue, or profit) to coordinates on the screen. You do not typically use the NumericAxis base class directly. Instead, you use the following subclasses when you define your axis:

- LinearAxis
- LogAxis
- DateTimeAxis

These classes give you significant control over how to set the appearance and values of elements along the axis such as labels and tick marks.

You can use the `baseAtZero` property to set the interval at which tick marks appear along the axis and to determine whether or not the axis starts at zero. Set this property to `true` to start the axis at zero; set it to `false` to let Flex determine a reasonable starting value relative to the values along the axis.

You can use the `parseFunction` property to specify a custom method that formats the data points in your chart. This property is supported by all subclasses of the NumericAxis class. For a detailed description of using this property with the DateTimeAxis, see "Using the parseFunction property" on page 1707.

If you want to change the values of the labels, use the `labelFunction` property of the NumericAxis class. For more information, see "Defining axis labels" on page 1715.

About the LinearAxis subclass

The LinearAxis subclass is the simplest of the three NumericAxis subclasses. It maps numeric values evenly between minimum and maximum values along a chart axis. By default, Flex determines the minimum, maximum, and interval values from the charting data to fit all of the chart elements on the screen. You can also explicitly set specific values for these properties. The following example sets the minimum and maximum values to 10 and 100, respectively:

```
<?xml version="1.0"?>
<!-- charts/LinearAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
  <![CDATA[
     import mx.collections.ArrayCollection;

    [Bindable]
    public var stocks:ArrayCollection = new ArrayCollection([
        {date:"2005/8/4", SMITH:37.23},
        {date:"2005/8/5", SMITH:56.53},
        {date:"2005/8/6", SMITH:17.67},
        {date:"2005/8/7", SMITH:27.72},
        {date:"2005/8/8", SMITH:85.23}
     ]);
  ]]>
  </mx:Script>

  <mx:LineChart id="mychart"
    dataProvider="{stocks}"
    showDataTips="true"
    height="300"
    width="400"
  >
     <mx:verticalAxis>
        <mx:LinearAxis
            title="linear axis"
            minimum="10"
            maximum="100"
            interval="10"
        />
     </mx:verticalAxis>

     <mx:horizontalAxis>
        <mx:CategoryAxis categoryField="date"/>
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries
            yField="SMITH"
            displayName="SMITH close"
        />
     </mx:series>
```

```
  </mx:LineChart>
</mx:Application>
```

About the LogAxis subclass

The LogAxis subclass is similar to the LinearAxis subclass, but it maps values to the axis logarithmically rather than linearly. You use a LogAxis object when the data in your chart has such a wide range that clusters of data points are lost to scale. LogAxis data also cannot be rendered if it is negative. For example, if you track the stock price of a successful company since 1929, it is useful to represent the data logarithmically rather than linearly so that the chart is readable.

When you use a LogAxis object, you set a multiplier that defines the values of the labels along the axis. You set the multiplier with the `interval` property. Values must be even powers of 10, and must be greater than or equal to 0. A value of 10 generates labels at 1, 10, 100, and 1000. A value of 100 generates labels at 1, 100, and 10,000. The default value of the `interval` property is 10. The LogAxis object rounds the interval to an even power of 10, if necessary.

As with the vertical and horizontal axes, you can also set the minimum and maximum values of a LogAxis object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LogAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
  <![CDATA[
     import mx.collections.ArrayCollection;

    [Bindable]
    public var stocks:ArrayCollection = new ArrayCollection([
        {date:"2005/8/4", SMITH:37.23, DECKER:1500},
        {date:"2005/8/5", SMITH:56.53, DECKER:1210},
        {date:"2005/8/6", SMITH:17.67, DECKER:1270},
        {date:"2005/8/7", SMITH:27.72, DECKER:1370},
        {date:"2005/8/8", SMITH:85.23, DECKER:1530}
     ]);
  ]]>
  </mx:Script>

  <mx:LineChart id="mychart"
    dataProvider="{stocks}"
    showDataTips="true"
    height="300"
    width="400"
  >
     <mx:verticalAxis>
        <mx:LogAxis title="log axis"
            interval="10"
            minimum="10"
            maximum="10000"
        />
     </mx:verticalAxis>

     <mx:horizontalAxis>
        <mx:CategoryAxis categoryField="date"/>
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries yField="DECKER" displayName="DECKER close"/>
        <mx:LineSeries yField="SMITH" displayName="SMITH close" />
     </mx:series>
  </mx:LineChart>
</mx:Application>
```

About the DateTimeAxis subclass

The DateTimeAxis subclass maps time-based values to a chart axis. The DateTimeAxis subclass calculates the minimum and maximum values to align with logical date and time units (for example, the nearest hour or the nearest week). The DateTimeAxis subclass also selects a time unit for the interval so that the chart renders a reasonable number of labels.

The `dataUnits` property of the DateTimeAxis subclass specifies how Flex should interpret the Date objects. Flex determines this property by default, but you can override it. To display data in terms of days, set the `dataUnits` property to *days*, as the following example shows:

```
<mx:DateTimeAxis dataUnits="days"/>
```

Valid values for the `dataUnits` property are `milliseconds`, `seconds`, `minutes`, `hours`, `days`, `weeks`, `months`, and `years`.

When assigning appropriate label units, a DateTimeAxis object does not assign any unit smaller than the units represented by the data. If the `dataUnits` property is set to `days`, the chart does not render labels for every hour, no matter what the minimum or maximum range is. To achieve this, you must set the value explicitly.

Some series use the value of the `dataUnits` property to affect their rendering. Specifically, most columnar series (such as Column, Bar, Candlestick, and HLOC controls) use the value of `dataUnits` to determine how wide to render their columns. If, for example, the ColumnChart control's horizontal axis has its `dataUnits` set to weeks and `dataUnits` set to days, the ColumnChart control renders each column at one-seventh the distance between labels.

About supported types

Data points on the DateTimeAxis object support the Date, String, and Number data types.

- Date: If the value of the data point is an instance of a Date object, it already represents an absolute date-time value and needs no interpretation. To pass a Date object as a data value, use the `parseFunction` property of the DateTimeAxis subclass. The `parseFunction` property returns a Date object. For more information, see "Using the parseFunction property" on page 1707.
- String: You can use any format that the `Date.parse()` method supports. The supported formats are:
  - *`Day Month DD Hours:Minutes:Seconds GMT Year`* (for example, Tue Feb 1 12:00:00 GMT-0800 2005)
  - *`Day Month DD YYYY Hours:Minutes:Seconds AM|PM`* (for example, Tue Feb 1 2005 12:00:00 AM)
  - *`Day Month DD YYYY`* (for example, Tue Feb 1 2005)

- *MM/DD/YYYY* (for example, 02/01/2005)

  You can also write custom logic that uses the `parseFunction` property of the DateTimeAxis to take any data type and return a Date. For more information, see "Using the parseFunction property" on page 1707.

- Number: If you use a number, it is assumed to be the number of milliseconds since Midnight, 1/1/1970; for example, 543387600000. To get this value on an existing Date object, use the Date object's `getTime()` method.

The following example uses a String value for the date that matches the MM/DD/YYYY pattern and specifies that the dates are displayed in units of days:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var aapl:ArrayCollection = new ArrayCollection([
        {date:"08/01/2005", close:42.71},
        {date:"08/02/2005", close:42.99},
        {date:"08/03/2005", close:42.65}
     ]);
  ]]></mx:Script>
  <mx:Panel title="DateTimeAxis" width="100%" height="100%">
     <mx:LineChart id="mychart"
        dataProvider="{aapl}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="close"
                xField="date"
                displayName="AAPL"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

Using the parseFunction property

You use the `parseFunction` property of the DateTimeAxis object to specify a method that customizes the value of the data points. With this property, you specify a method that accepts a value and returns a Date object. The Date object is then used in the DateTimeAxis object of the chart. This lets you provide customizable date input strings and convert them to Date objects, which Flex can then interpret for use in the DateTimeAxis.

The parsing method specified by the `parseFunction` property is called every time a value for the DateTimeAxis must be calculated. It is called each time a data point is encountered when the user interacts with the chart. Consequently, Flex might call the parsing method often, which can degrade an application's performance. Therefore, you should try to keep the amount of code in the parsing method to a minimum.

Flex passes only one parameter to the parsing method. This parameter is the value of the data point you specified for the series. Typically, it is a string representing some form of a date. You cannot override this parameter or add additional parameters.

The following example shows a parsing method that creates a Date object from String values in the data provider that match the "YYYY, MM, DD" pattern:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisParseFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var aapl:ArrayCollection = new ArrayCollection([
        {date:"2005, 8, 1", close:42.71},
        {date:"2005, 8, 2", close:42.99},
        {date:"2005, 8, 3", close:44}
     ]);

     public function myParseFunction(s:String):Date {
        // Get an array of Strings from the
        // comma-separated String passed in.
        var a:Array = s.split(",");

        // Trace out year, month, and day values.
        trace("y:" + a[0]);
        trace("m:" + a[1]);
        trace("d:" + a[2]);

        // Create the new Date object.
        var newDate:Date = new Date(a[0],a[1],a[2]);
        return newDate;
     }
  ]]></mx:Script>
  <mx:LineChart id="mychart"
    dataProvider="{aapl}"
    showDataTips="true"
  >
     <mx:horizontalAxis>
        <mx:DateTimeAxis
            dataUnits="days"
            parseFunction="myParseFunction"
        />
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries
            yField="close"
            xField="date"
            displayName="AAPL"
        />
     </mx:series>
  </mx:LineChart>
</mx:Application>
```

Formatting DateTimeAxis labels

When assigning the units to display along the axis, the DateTimeAxis object uses the largest unit allowed to render a reasonable number of labels. The following table describes the default label format and the minimum range for each unit type:

Unit	Label format	Minimum range
Years	YYYY	If the minimum and maximum values span at least 2 years.
Months	MM/YY	Spans at least 2 months.
Weeks	DD/MM/YY	Spans at least 2 weeks.
Days	DD/MM/YY	Spans at least 1 day.
Hours	HH:MM	Spans at least 1 hour.
Minutes	HH:MM	Spans at least 1 minute.
Seconds	HH:MM:SS	Spans at least 1 second.
Milliseconds	HH:MM:SS:mmmm	Spans at least 1 millisecond.

You can restrict the list of valid units for a particular chart instance to a subset that makes sense for the use case. As with a LinearAxis object, you can specify minimum, maximum, and interval values for a DateTimeAxis object.

When rounding off values, the DateTimeAxis object determines if values passed to it should be displayed in the local time zone or UTC. You can set the `displayLocalTime` property to `true` to instruct the DateTimeAxis object to treat values as local time values. The default value is `false`.

To change the values of the labels, use the `labelFunction` property of the DateTimeAxis object. This property is inherited from the NumericAxis class and is described in "Defining axis labels" on page 1715.

Setting minimum and maximum values on a DateTimeAxis

You can define the range of values that any axis uses by setting the values of the `minimum` and `maximum` properties on that axis. For the DateTimeAxis class, however, you must use Date objects and not Numbers or Strings to define that range. To do this, you create bindable Date objects and bind the values of the `minimum` and `maximum` properties to those objects, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisRange.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var minDate:Date = new Date(2005, 11, 1);

     [Bindable]
     public var maxDate:Date = new Date(2007, 1, 1);

     [Bindable] public var myData:ArrayCollection = new
     ArrayCollection([
        {date: "01/01/2006", amt: 12345},
        {date: "02/01/2006", amt: 54331},
        {date: "12/31/2006", amt: 34343}
     ]);

  ]]></mx:Script>
  <mx:Panel title="DateTimeAxis" width="100%" height="100%">
     <mx:PlotChart id="mychart"
        dataProvider="{myData}"
        showDataTips="true">
        <mx:horizontalAxis>
           <mx:DateTimeAxis
                dataUnits="months"
                minimum="{minDate}"
                maximum="{maxDate}"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:PlotSeries
                yField="amt"
                xField="date"
                displayName="myData"
           />
        </mx:series>
     </mx:PlotChart>
  </mx:Panel>
</mx:Application>
```

You can also represent the range of dates in MXML by using the following syntax:

```
<mx:horizontalAxis>
  <mx:DateTimeAxis dataUnits="months">
    <mx:minimum>
      <mx:Date fullYear="2005" month="11" date="1"/>
    </mx:minimum>
    <mx:maximum>
      <mx:Date fullYear="2007" month="1" date="1"/>
    </mx:maximum>
  </mx:DateTimeAxis>
</mx:horizontalAxis>
```

Adding axis titles

Each axis in a chart control can include a title that describes the purpose of the axis to the users. Flex does not add titles to the chart's axes unless you explicitly set them. To add titles to the axes of a chart, you use the `title` property of the axis object. This is CategoryAxis or one of the NumericAxis subclasses such as DateTimeAxis, LinearAxis, or LogAxis. To set a style for the axis title, use the `axisTitleStyleName` property of the chart control.

The following example sets the titles of the horizontal and vertical axes (in MXML and ActionScript), and applies the styles to those titles:

```
<?xml version="1.0"?>
<!-- charts/AxisTitles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="setTitles()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     private function setTitles():void {
        la1.title="Dollars";
     }

  ]]></mx:Script>

  <mx:Style>
     .myStyle {
        fontFamily:Verdana;
        fontSize:12;
        color:#4691E1;
        fontWeight:bold;
```

```
        fontStyle:italic;
    }
  </mx:Style>

  <mx:Panel>
    <mx:ColumnChart id="myChart"
      axisTitleStyleName="myStyle"
      dataProvider="{expenses}"
    >
      <mx:verticalAxis>
        <mx:LinearAxis id="la1"/>
      </mx:verticalAxis>
      <mx:horizontalAxis>
        <mx:CategoryAxis title="FY 2006"
            categoryField="Month"
        />
      </mx:horizontalAxis>
      <mx:series>
        <mx:ColumnSeries
            xField="Month"
            yField="Profit"
            displayName="Profit"
        />
        <mx:ColumnSeries
            xField="Month"
            yField="Expenses"
            displayName="Expenses"
        />
      </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use embedded fonts for your axis titles. The following example embeds the font and sets the style for the vertical axis title:

```
<?xml version="1.0"?>
<!-- charts/AxisTitleEmbedFont.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>

  <mx:Style>
```

```
    @font-face{
       src:url("../assets/MyriadWebPro.ttf");
       fontFamily:myMyriad;
    }

    .myEmbeddedStyle {
       fontFamily:myMyriad;
       fontSize:20;
    }
  </mx:Style>

  <mx:Panel>
    <mx:ColumnChart id="column"
       dataProvider="{expenses}"
       axisTitleStyleName="myEmbeddedStyle"
    >
       <mx:horizontalAxis>
          <mx:CategoryAxis
               dataProvider="{expenses}"
               categoryField="Month"
               title="FY 2006"
          />
       </mx:horizontalAxis>
       <mx:series>
          <mx:ColumnSeries
               xField="Month"
               yField="Profit"
               displayName="Profit"
          />
          <mx:ColumnSeries
               xField="Month"
               yField="Expenses"
               displayName="Expenses"
          />
       </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

For information on embedding fonts, see "Using embedded fonts" on page 767.

You can take advantage of the fact that the chart applies the `axisTitleStyleName` property without explicitly specifying it, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CSSAxisTitle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>

  <mx:Style>
     .axisTitles {
        color:red;
        fontWeight:bold;
        fontFamily:Arial;
        fontSize:20;
     }

     ColumnChart {
        axisTitleStyleName:axisTitles;
     }
  </mx:Style>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month" title="FY 2006"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
```

```
    <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

You can also apply the title style to the axis, as the following example shows:

```
<mx:CategoryAxis title="State" styleName="myEmbeddedStyle"/>
```

Defining axis labels

You define the values of axis labels on the horizontal axis or vertical axis.

You can disable labels by setting the value of the `showLabels` property to `false` on the AxisRenderer object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/DisabledAxisLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderer>
           <mx:AxisRenderer showLabels="false"/>
        </mx:horizontalAxisRenderer>
        <mx:verticalAxisRenderer>
           <mx:AxisRenderer showLabels="false"/>
        </mx:verticalAxisRenderer>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
```

```
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

You can customize the value of axis labels by using the `labelFunction` callback function of the axis. The function specified in `labelFunction` returns a String that Flex displays as the axis label.

The required format of the callback function is:

```
function function_name(labelValue:Object, previousLabelValue:Object,
  axis:axis_type, labelItem:Object):return_type
```

The following table describes the parameters of the callback function:

Parameter	Description
labelValue	The value of the current label.
previousLabelValue	The value of the label preceding this label. If this is the first label, the value of *previousLabelValue* is `null`.
axis	The axis object, such as CategoryAxis or NumericAxis.
labelItem	A reference to the label object.
return_type	The type of object that the callback function returns. This can be any object type, but is most commonly a String for CategoryAxis axes, a Number for NumericAxis objects, or a Date object for DateTimeAxis objects.

When you use the `labelFunction`, you must be sure to import the class of the axis or the entire charts package; for example:

```
import mx.charts.*;
```

The following example defines a `labelFunction` for the horizontal CategoryAxis object. In that function, Flex appends '06 to the axis labels, and displays the labels as Jan '06, Feb '06, and Mar '06. The return type of the label formatting function is a String because it uses the month name and appends the year to that String.

```
<?xml version="1.0"?>
<!-- charts/CustomLabelFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.charts.*;
  import mx.collections.ArrayCollection;
```

```
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month: "Jan", Profit: 2000, Expenses: 1500},
     {Month: "Feb", Profit: 1000, Expenses: 200},
     {Month: "Mar", Profit: 1500, Expenses: 500}
  ]);

  public function defineLabel(
    cat:Object,
    pcat:Object,
    ax:CategoryAxis,
    catItem:Object):String
  {
        return cat + " '06";
  }
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                title="Expenses"
                labelFunction="defineLabel"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

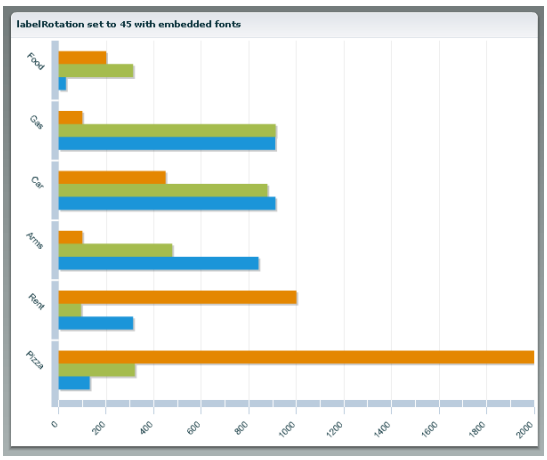
Another way to customize the labels on an axis is to set a custom data provider for the labels. For more information, see "About the CategoryAxis class" on page 1698.

You can customize labels for the PieChart control with the label function defined on the PieSeries class. For more information, see "Using labels with PieChart controls" on page 1658.

Rotating axis elements

You can rotate axis labels using the `labelRotation` property of the AxisRenderer object. You specify a number from -90 to 90, in degrees. If you set the `labelRotation` property to `null`, Flex determines an optimal angle and renders the labels.

The following example shows both sets of axis labels rotated 45 degrees:



To rotate text, you must embed the font in the Flex application. The following `<mx:Style>` block embeds the Arial font in the Flex application, and then applies that font to the chart control that rotates its labels 45 degrees:

```
<?xml version="1.0"?>
<!-- charts/RotateAxisLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month: "Jan", Profit: 2000, Expenses: 1500},
     {Month: "Feb", Profit: 1000, Expenses: 200},
     {Month: "Mar", Profit: 1500, Expenses: 500}
  ]);
  ]]></mx:Script>

  <mx:Style>
     @font-face{
        src: url("../assets/MyriadWebPro.ttf");
        fontFamily: myMyriad;
     }

     ColumnChart {
```

```
        fontFamily: myMyriad;
        fontSize: 20;
     }
  </mx:Style>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                title="FY 2006"
           />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderer>
           <mx:AxisRenderer labelRotation="45"/>
        </mx:horizontalAxisRenderer>
        <mx:verticalAxisRenderer>
           <mx:AxisRenderer labelRotation="45"/>
        </mx:verticalAxisRenderer>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
               displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
               displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

Setting ranges

Flex determines the minimum and maximum values along an axis and sets the interval based on the settings of the NumericAxis object. You can override the values that Flex calculates. By changing the range of the data displayed in the chart, you also change the range of the tick marks.

The following table describes the properties of the axis that define the ranges along the axes:

Property	Description
`minimum`	The lowest value of the axis.
`maximum`	The highest value of the axis.
`interval`	The number of units between values along the axis.

The following example defines the range of the y-axis:

```
<?xml version="1.0"?>
<!-- charts/LinearAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
  <![CDATA[
     import mx.collections.ArrayCollection;

    [Bindable]
    public var stocks:ArrayCollection = new ArrayCollection([
        {date:"2005/8/4", SMITH:37.23},
        {date:"2005/8/5", SMITH:56.53},
        {date:"2005/8/6", SMITH:17.67},
        {date:"2005/8/7", SMITH:27.72},
        {date:"2005/8/8", SMITH:85.23}
     ]);
  ]]>
  </mx:Script>

  <mx:LineChart id="mychart"
    dataProvider="{stocks}"
    showDataTips="true"
    height="300"
    width="400"
  >
     <mx:verticalAxis>
        <mx:LinearAxis
            title="linear axis"
            minimum="10"
            maximum="100"
            interval="10"
        />
     </mx:verticalAxis>
```

```
    <mx:horizontalAxis>
       <mx:CategoryAxis categoryField="date"/>
    </mx:horizontalAxis>
    <mx:series>
       <mx:LineSeries
           yField="SMITH"
           displayName="SMITH close"
       />
    </mx:series>
  </mx:LineChart>
</mx:Application>
```

In this example, the minimum value displayed along the y-axis is 10, the maximum value is 100, and the interval is 10. Therefore, the label text is 10, 20, 30, 40, and so on.

To set the minimum and maximum values on a DateTimeAxis, you must use Date objects rather than Strings or Numbers in the axis's tag. For more information, see "Setting minimum and maximum values on a DateTimeAxis" on page 1710.

For information about setting the length and location of tick marks, see "Formatting tick marks" on page 1721.

Formatting tick marks

There are two types of tick marks on a Flex chart: major and minor. Major tick marks are the indications along an axis that correspond to an axis label. The text for the axis labels is often derived from the chart's data provider. Minor tick marks are those tick marks that appear between the major tick marks. Minor tick marks help the user visualize the distance between the major tick marks.

You use the `tickPlacement` and `minorTickPlacement` properties of the AxisRenderer object to determine whether or not Flex displays tick marks and where Flex displays tick marks.

The following table describes valid values of the `tickPlacement` and `minorTickPlacement` properties:

Value	Description
`cross`	Places tick marks across the axis.
`inside`	Places tick marks on the inside of the axis line.
`none`	Hides tick marks.
`outside`	Places tick marks on the outside of the axis line.

You can align the tick marks with labels by using the `tickAlignment` property.

Flex also lets you set the length of tick marks and the number of minor tick marks that appear along the axis. The following table describes the properties that define the length of tick marks on the chart's axes:

Property	Description
`tickLength`	The length, in pixels, of the major tick mark from the axis.
`minorTickLength`	The length, in pixels, of the minor tick mark from the axis.

The minor tick marks overlap the placement of major tick marks. So, if you hide major tick marks but still show minor tick marks, the minor tick marks appear at the regular tick-mark intervals.

The following example sets tick marks to the inside of the axis line, sets the tick mark's length to 12 pixels, and hides minor tick marks:

```
<?xml version="1.0"?>
<!-- charts/TickStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  [Bindable]
  public var aapl:Array = [
     {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
     {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
     {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,
     close:43.22},
     {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
     {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,
     close:42.99},
     {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
     {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,
     close:43.82},
     {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
     {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
     {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,
     close:46.1},
  ];
  ]]></mx:Script>

  <mx:Style>
     .myAxisStyle {
        placement:bottom;
        minorTickPlacement:none;
        tickLength:12;
        tickPlacement:inside;
     }
  </mx:Style>

    <mx:HLOCChart id="mychart"
```

```
        dataProvider="{aapl}"
        showDataTips="true"
    >
        <mx:horizontalAxisRenderer>
        <mx:AxisRenderer styleName="myAxisStyle"/>
    </mx:horizontalAxisRenderer>

    <mx:verticalAxis>
        <mx:LinearAxis minimum="30" maximum="50"/>
    </mx:verticalAxis>

    <mx:series>
        <mx:HLOCSeries
            dataProvider="{aapl}"
            openField="open"
            highField="high"
            lowField="low"
            closeField="close"
            displayName="AAPL"
        />
     </mx:series>
  </mx:HLOCChart>
  <mx:Legend dataProvider="{mychart}"/>
</mx:Application>
```

Formatting axis lines

Axes have lines to which the tick marks are attached. You can use style properties to hide these lines or change the width of the lines.

To hide the axis line, set the value of the `showLine` property on the AxisRenderer object to `false`. The default value is `true`. The following example sets `showLine` to `false`:

```
<?xml version="1.0"?>
<!-- charts/DisableAxisLines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart dataProvider="{expenses}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderer>
           <mx:AxisRenderer showLine="false"/>
        </mx:horizontalAxisRenderer>
        <mx:verticalAxisRenderer>
           <mx:AxisRenderer showLine="false"/>
        </mx:verticalAxisRenderer>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also apply the `showLine` property as a CSS style property.

You can change the `width`, `color`, and `alpha` of the axis line with the `<mx:axisStroke>` tag. You use an `<mx:Stroke>` child tag to define these properties or define a stroke and then bind it to the `axisStroke` object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/StyleAxisLines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>

  <mx:Stroke id="axisStroke"
        color="#884422"
        weight="8"
        alpha=".75"
        caps="square"
  />

  <mx:Panel title="Line Chart">
     <mx:LineChart dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderer>
           <mx:AxisRenderer>
            <mx:axisStroke>{axisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:horizontalAxisRenderer>
        <mx:verticalAxisRenderer>
           <mx:AxisRenderer>
            <mx:axisStroke>
                <mx:Stroke color="#884422"
                    weight="8"
                    alpha=".75"
                    caps="square"
                />
            </mx:axisStroke>
           </mx:AxisRenderer>
        </mx:verticalAxisRenderer>
```

```
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

For more information about strokes, see "Using strokes" on page 1728.

You can also apply filters to axis lines to further customize their appearance. For more information, see "Using filters" on page 1747.

Using the minField property

Some series types let you specify a minimum value for the elements drawn on the screen. For example, in a ColumnChart control, you can specify the base value of the column. To specify a base value (or minimum) for the column, set the value of the series object's `minField` property to the data provider field.

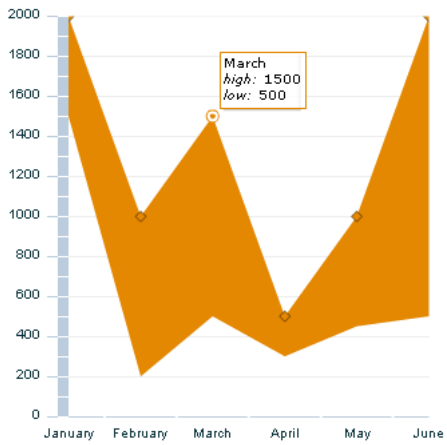
You can specify the `minField` property for the following chart series types:

- AreaSeries
- BarSeries
- ColumnSeries

Setting a value for the `minField` property creates two values on the axis for each data point in an area; for example:

```
<?xml version="1.0"?>
<!-- charts/MinField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                minField="Expenses"
                displayName="Profit"
            />
        </mx:series>
     </mx:AreaChart>
  </mx:Panel>
</mx:Application>
```

The resulting DataTip labels the current value “high” and the `minField` value “low.” The following example shows an AreaChart that defines the base of each column:



For an example of using the `minField` property to create a waterfall or cascading ColumnChart control, see “Using column charts” on page 1635.

Using strokes

You use the Stroke class with the chart series and grid lines to control the properties of the lines that Flex uses to draw chart elements.

The following table describes the properties that you use to control the appearance of strokes:

Property	Description
`color`	Specifies the color of the line as a hexadecimal value. The default value is `0x000000`, which corresponds to black.
`weight`	Specifies the width of the line, in pixels. The default value is 0, which corresponds to a hairline.
`alpha`	Specifies the transparency of a line. Valid values are 0 (invisible) through 100 (opaque). The default value is 100.

The following example defines a line width of 2 pixels, one with a dark gray border (0x808080) and the other with a light gray border (0xC0C0C0) for the borders of items in a BarChart control's BarSeries:

```
<?xml version="1.0"?>
<!-- charts/BasicBarStroke.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
            >
            <mx:stroke>
                <mx:Stroke
                    color="0x808080"
                    weight="2"
                    alpha=".8"
                />
            </mx:stroke>
           </mx:BarSeries>
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           >
            <mx:stroke>
                <mx:Stroke
                    color="0xC0C0C0"
                    weight="2"
                    alpha=".8"
                />
            </mx:stroke>
```

```
          </mx:BarSeries>
        </mx:series>
    </mx:BarChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Defining AxisRenderer properties with strokes

You can use strokes to define tick marks and other properties of an AxisRenderer, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/AxisRendererStrokes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  [Bindable]
  public var aapl:Array = [
     {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
     {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
     {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
     {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
     {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
     {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
     {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
     {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
     {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
     {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1},
  ];
  ]]></mx:Script>

  <mx:HLOCChart id="myChart"
    dataProvider="{aapl}"
    showDataTips="true"
  >
     <mx:horizontalAxisRenderer>
        <mx:AxisRenderer
            placement="bottom"
            canDropLabels="true"
            tickPlacement="inside"
            tickLength="10"
            minorTickPlacement="inside"
            minorTickLength="5"
        >
           <mx:axisStroke>
            <mx:Stroke color="#000080" weight="1"/>
           </mx:axisStroke>
           <mx:tickStroke>
            <mx:Stroke color="#000060" weight="1"/>
```

```
            </mx:tickStroke>
            <mx:minorTickStroke>
             <mx:Stroke color="#100040" weight="1"/>
            </mx:minorTickStroke>
         </mx:AxisRenderer>
      </mx:horizontalAxisRenderer>

      <mx:verticalAxis>
         <mx:LinearAxis minimum="30" maximum="50"/>
      </mx:verticalAxis>

      <mx:series>
         <mx:HLOCSeries
             dataProvider="{aapl}"
             openField="open"
             highField="high"
             lowField="low"
             closeField="close"
             displayName="AAPL"
         >
         </mx:HLOCSeries>
      </mx:series>
   </mx:HLOCChart>
   <mx:Legend dataProvider="{myChart}"/>
</mx:Application>
```

You can define a stroke object using an MXML tag, and then bind that stroke object to the chart's renderer properties. For an example, see "Applying styles by binding tag definitions" on page 1689.

Using strokes in ActionScript

You can instantiate and manipulate a Stroke object in ActionScript by using the mx.graphics.Stroke class. You can then use the `setStyle()` method to apply the Stroke object to the chart, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ActionScriptStroke.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.graphics.Stroke;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2000, Expenses: 1500},
        {Month: "Feb", Profit: 1000, Expenses: 200},
        {Month: "Mar", Profit: 1500, Expenses: 500}
     ]);

     public function changeStroke(e:Event):void {
        var s:Stroke = new Stroke(0x001100,2);
        s.alpha = .5;
        s.color = 0x0000FF;
        har1.setStyle("axisStroke",s);
        var1.setStyle("axisStroke",s);
     }
  ]]></mx:Script>

  <mx:Stroke id="baseAxisStroke"
    color="0x884422"
    weight="10"
    alpha=".25"
    caps="square"
  />

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderer>
           <mx:AxisRenderer id="har1">
            <mx:axisStroke>{baseAxisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:horizontalAxisRenderer>
```

```
        <mx:verticalAxisRenderer>
           <mx:AxisRenderer id="var1">
            <mx:axisStroke>{baseAxisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:verticalAxisRenderer>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
  <mx:Button id="b1"
    click="changeStroke(event)"
    label="Change Stroke"
  />
</mx:Application>
```

Defining strokes for LineSeries and AreaSeries

Some chart series have more than one stroke-related style property. For LineSeries, you use the `stroke` style property to define a style for the chart item's renderer. You use the `lineStroke` property to define the stroke of the actual line segments.

The following example creates a thick blue line for the LineChart control's line segments, with large red boxes at each data point, which use the CrossItemRenderer object as their renderer:

```
<?xml version="1.0"?>
<!-- charts/LineSeriesStrokes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
            yField="Profit"
            displayName="Profit"
           >
            <mx:itemRenderer>
                <mx:Component>
                    <mx:CrossItemRenderer
                        scaleX="1.5"
                        scaleY="1.5"
                    />
                </mx:Component>
            </mx:itemRenderer>
            <mx:fill>
                <mx:SolidColor
                    color="0x0000FF"
                />
            </mx:fill>
            <mx:stroke>
                <mx:Stroke
                    color="0xFF0066"
                    alpha="1"
                />
            </mx:stroke>
```

```
            <mx:lineStroke>
                <mx:Stroke
                    color="0x33FFFF"
                    weight="5"
                    alpha=".8"
                />
            </mx:lineStroke>
          </mx:LineSeries>
          <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
          />
        </mx:series>
    </mx:LineChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Similarly, with the AreaSeries class, you use the `stroke` property to set a style for the chart item's renderer. You use the `areaStroke` style property to define the stroke of the line that defines the area.

Using fills

When charting multiple data series, or just to improve the appearance of your charts, you can control the fill for each series in the chart. The fill lets you specify a pattern that defines how Flex draws the chart element. You can also use fills to specify the background colors of the chart or bands of background colors defined by the grid lines. Fills can be solid or can use linear and radial gradients. A *gradient* specifies a gradual color transition in the fill color

You use the `fill` property of the chart series to define the characteristics of the fill. The LineSeries and LineRenderer objects are not affected by the `fill` property's settings. The PieSeries object supports an array of `fill`s rather than a single `fill` property.

One of the most common uses of a fill is to control the color of the chart when you have multiple data series in a chart. The following example uses the `fill` property to set the color for each ColumnSeries object in a ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/ColumnFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           >
               <mx:fill>
                    <mx:SolidColor color="0x336699"/>
               </mx:fill>
           </mx:ColumnSeries>

           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           >
            <mx:fill>
                <mx:SolidColor color="0xFF99FF"/>
            </mx:fill>
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If you do not explicitly define different fills for multiple data series, Flex chooses solid colors for you.

With the PieSeries, you can use an array of fills to specify how Flex should draw the individual wedges. For example, you can give each wedge that represents a PieSeries its own color, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/PieFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
    ]]>
  </mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           >
                <mx:fills>
                    <mx:SolidColor color="0xCC66FF" alpha=".8"/>
                    <mx:SolidColor color="0x9966CC" alpha=".8"/>
                    <mx:SolidColor color="0x9999CC" alpha=".8"/>
                    <mx:SolidColor color="0x6699CC" alpha=".8"/>
                    <mx:SolidColor color="0x669999" alpha=".8"/>
                    <mx:SolidColor color="0x99CC99" alpha=".8"/>
                </mx:fills>
            </mx:PieSeries>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use fills to set the background of the charts. You do this by adding an `<mx:fill>` child tag to the chart tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BackgroundFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Background Fill">
     <mx:BarChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:fill>
           <mx:SolidColor
                color="0x66CCFF"
                alpha=".5"
           />
        </mx:fill>
        <mx:series>
           <mx:BarSeries xField="Amount"/>
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
</mx:Application>
```

Setting fills with CSS

You can use the `fill` property in an `<mx:Style>` declaration using CSS syntax. You can use either a type or class selector. The following example sets the fill of the custom `myBarChartStyle` class selector to `#FF0000`:

```
<?xml version="1.0"?>
<!-- charts/BackgroundFillsCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1100},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>

  <mx:Style>
     .myBarChartStyle {
        fill:#FF0000;
     }
  </mx:Style>

  <mx:Panel title="Background Fill">
     <mx:BarChart
        dataProvider="{expenses}"
        showDataTips="true"
        styleName="myBarChartStyle"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries xField="Amount"/>
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
</mx:Application>
```

Flex converts this to a SolidColor object.

Using a gradient fill with chart controls

Flex provides two fill classes that let you specify a gradient fill. You use either the LinearGradient class or the RadialGradient class, along with the GradientEntry class to specify a gradient fill. The following table describes these classes:

Class	Description
LinearGradient	Defines a gradient fill that starts at a boundary of the chart element. You specify an array of GradientEntry objects to control gradient transitions with the LinearGradient object.
RadialGradient	Defines a gradient fill that radiates from the center of a chart element. You specify an array of GradientEntry objects to control gradient transitions with the RadialGradient object.
GradientEntry	Defines the objects that control the gradient transition. Each GradientEntry object contains the following properties: • `color` Specifies a color value. • `alpha` Specifies the transparency. Valid values are 0 (invisible) through 1 (opaque). The default value is 1. • `ratio` Specifies where in the chart, as a percentage, Flex starts the transition to the next `color`. For example, if you set the `ratio` property to .33, Flex begins the transition 33% of the way through the chart. If you do not set the `ratio` property, Flex tries to evenly apply values based on the `ratio` properties for the other GradientEntry objects. Valid values range from 0 to 1.

The following example uses a LinearGradient class with three colors for a gradient fill of the chart's data series:

```
<?xml version="1.0"?>
<!-- charts/GradientFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Background Fill">
     <mx:BarChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
```

```
            <mx:CategoryAxis
                 dataProvider="{expenses}"
                 categoryField="Expense"
            />
         </mx:verticalAxis>
         <mx:fill>
            <mx:LinearGradient>
             <mx:entries>
                 <mx:GradientEntry
                     color="0xC5C551"
                     ratio="0"
                     alpha="1"
                 />
                 <mx:GradientEntry
                     color="0xFEFE24"
                     ratio=".33"
                     alpha="1"
                 />
                 <mx:GradientEntry
                     color="0xECEC21"
                     ratio=".66"
                     alpha="1"
                 />
             </mx:entries>
            </mx:LinearGradient>
         </mx:fill>
         <mx:series>
            <mx:BarSeries xField="Amount"/>
         </mx:series>
      </mx:BarChart>
   </mx:Panel>
</mx:Application>
```

The LinearGradient object takes a single attribute, `angle`. By default, it defines a transition from left to right across the chart. Use the `angle` property to control the direction of the transition. For example, a value of 180 causes the transition to occur from right to left, rather than from left to right.

The following example sets the `angle` property to 90, which specifies that the transition occurs from the top of the chart to the bottom.

```
<?xml version="1.0"?>
<!-- charts/GradientFillsAngled.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Background Fill">
     <mx:BarChart dataProvider="{expenses}" showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:fill>
           <mx:LinearGradient angle="90">
            <mx:entries>
                <mx:GradientEntry
                    color="0xC5C551"
                    ratio="0"
                    alpha="1"
                />
                <mx:GradientEntry
                    color="0xFEFE24"
                    ratio=".33"
                    alpha="1"
                />
                <mx:GradientEntry
                    color="0xECEC21"
                    ratio=".66"
                    alpha="1"
                />
            </mx:entries>
           </mx:LinearGradient>
        </mx:fill>
        <mx:series>
           <mx:BarSeries xField="Amount"/>
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
```

```
</mx:Application>
```

Using different alpha values with a fill

When charting multiple data series, you can define the series to overlap. For example, the column chart lets you display the columns next to each other or overlap them for multiple data series. The same is true for an area series.

When you have multiple data series that overlap, you can specify that the fill for each series has an alpha value less than 100%, so that the series have a level of transparency. The valid values for the `alpha` property are 0 (invisible) through 1 (opaque).

The following example defines an area chart in which each series in the chart uses a solid fill with the same level of transparency:

```
<?xml version="1.0"?>
<!-- charts/AlphaFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           >
                <mx:areaStroke>
                    <mx:Stroke
                        color="0x9A9A00"
                        weight="2"
                    />
                </mx:areaStroke>
```

```
                <mx:areaFill>
                    <mx:SolidColor
                        color="0x7EAEFF"
                        alpha=".3"
                    />
                </mx:areaFill>
            </mx:AreaSeries>
            <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
            >
                <mx:areaStroke>
                    <mx:Stroke
                        color="0x9A9A00"
                        weight="2"
                    />
                </mx:areaStroke>
                <mx:areaFill>
                    <mx:SolidColor
                        color="0xAA0000"
                        alpha=".3"
                    />
                </mx:areaFill>
            </mx:AreaSeries>
        </mx:series>
    </mx:AreaChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If you define a gradient fill, you can set the `alpha` property on each entry in the Array of `<mx:GradientEntry>` tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GradientAlphaFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries yField="Profit" displayName="Profit">
            <mx:areaStroke>
                <mx:Stroke color="0x9A9A00" weight="2"/>
            </mx:areaStroke>
            <mx:areaFill>
                <mx:LinearGradient angle="90">
                    <mx:entries>
                        <mx:GradientEntry
                            color="0xC5C551"
                            ratio="0"
                            alpha="1"
                        />
                        <mx:GradientEntry
                            color="0xFEFE24"
                            ratio=".33"
                            alpha="1"
                        />
                        <mx:GradientEntry
                            color="0xECEC21"
                            ratio=".66"
                            alpha=".2"
                        />
                    </mx:entries>
                </mx:LinearGradient>
            </mx:areaFill>
```

```
            </mx:AreaSeries>
            <mx:AreaSeries
             yField="Expenses"
             displayName="Expenses"
            >
             <mx:areaStroke>
                 <mx:Stroke color="0x9A9A00" weight="2"/>
             </mx:areaStroke>
             <mx:areaFill>
                 <mx:LinearGradient angle="90">
                     <mx:entries>
                         <mx:GradientEntry
                             color="0xAA0000"
                             ratio="0"
                             alpha="1"
                         />
                         <mx:GradientEntry
                             color="0xCC0000"
                             ratio=".33"
                             alpha="1"
                         />
                         <mx:GradientEntry
                             color="0xFF0000"
                             ratio=".66"
                             alpha=".2"
                         />
                     </mx:entries>
                 </mx:LinearGradient>
             </mx:areaFill>
            </mx:AreaSeries>
        </mx:series>
    </mx:AreaChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

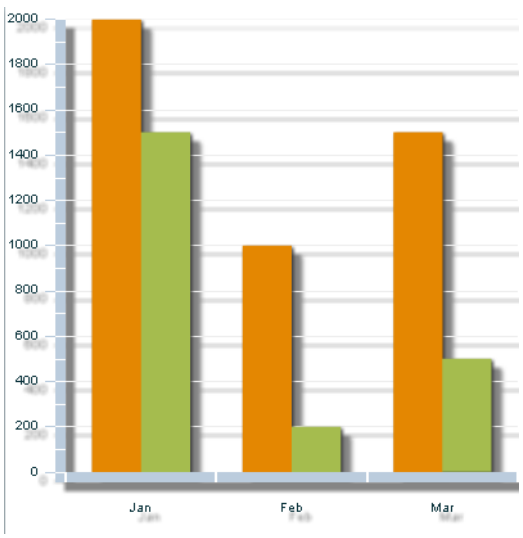
In this example, you make the last gradient color in the series fully transparent by setting its `alpha` property to 0.

Using filters

You can add filters such as drop shadows to your charts by using the classes in the flash.filters package. These filters include:

- BevelFilter
- BitmapFilter
- BlurFilter
- ColorMatrixFilter
- DisplacementMapFilter
- DropShadowFilter
- GlowFilter
- GradientBevelFilter
- GradientGlowFilter

You can apply filters to the chart control itself, or to each chart series. When you apply a filter to a chart control, the filter is applied to all aspects of that chart control, including gridlines, axis labels, and each data point in the series. The following image shows a drop shadow filter applied to a ColumnChart control:



The following example applies a custom drop shadow filter to a ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/ColumnWithDropShadow.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:flash="flash.filters.*">

  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <!-- Add a custom drop shadow filter to the
                ColumnChart control. -->
        <mx:filters>
           <flash:DropShadowFilter
                distance="10"
                color="0x666666"
                alpha=".8"
           />
        </mx:filters>
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

When you use a filter in your Flex application, ensure that you add the flash.filters namespace to your MXML file's top-level tag, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:flash="flash.filters.*">
```

For more information on using filters, see "Using filters in Flex" on page 753.

Adding a drop shadow filter to some chart controls can have unexpected consequences. For example, if you add a drop shadow filter to a PieChart control, Flex renders that drop shadow filter in addition to the drop shadow filter that it applies to the PieSeries by default.

You can remove filters by setting the filters Array to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ClearFilters.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
     {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
     {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
  ]);
  ]]></mx:Script>
  <mx:LineChart id="myChart" dataProvider="{expenses}">
     <mx:seriesFilters>
        <mx:Array/>
     </mx:seriesFilters>
     <mx:horizontalAxis>
        <mx:CategoryAxis
            dataProvider="{expenses}"
            categoryField="Month"
        />
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries
            yField="Profit"
            displayName="Profit"
        />
        <mx:LineSeries
            yField="Expenses"
            displayName="Expenses"
        />
        <mx:LineSeries
            yField="Amount"
            displayName="Amount"
        />
     </mx:series>
```

```
  </mx:LineChart>
</mx:Application>
```

The following example creates a PieChart control and applies a drop shadow to it; it also removes the default drop shadow filter from the PieSeries so that there is a single drop shadow:

```
<?xml version="1.0"?>
<!-- charts/PieChartShadow.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:flash=
"flash.filters.*">
  <mx:Script>
    <![CDATA[
     [Bindable]
     public var expenses:Object = [
        {Expense:"Taxes", Amount:2000},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ];
    ]]>
  </mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <!-- Add a custom drop shadow to the PieChart control -->
        <mx:filters>
           <flash:DropShadowFilter
                distance="10"
                color="0x666666"
                alpha=".8"
           />
        </mx:filters>
        <mx:series>
           <mx:Array>
            <mx:PieSeries field="Amount" nameField="Expense"
            labelPosition="callout" explodeRadius=".2">
                <!-- Clear default shadow on the PieSeries -->
                <mx:filters>
                    <mx:Array/>
                </mx:filters>
            </mx:PieSeries>
           </mx:Array>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

For more information on working with grid lines, see "Adding grid lines" on page 1753.

You can add filters to individual chart elements that are display objects, such as series, grid lines, legend items, and axes. The following example defines set of filters, and then applies them to various chart elements:

```
<?xml version="1.0"?>
<!-- charts/MultipleFilters.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:flash=
"flash.filters.*" creationComplete="createFilters()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     private var myBlurFilter:BlurFilter;
     private var myGlowFilter:GlowFilter;
     private var myBevelFilter:BevelFilter;
     private var myDropShadowFilter:DropShadowFilter;

     private var color:Number = 0xFF33FF;

     public function applyFilters():void {
        // Apply filters to series, grid lines, legend, and axis.
        myGridlines.filters = [myBlurFilter];
        myLegend.filters = [myGlowFilter];
        myAxisRenderer.filters = [myBevelFilter];
        s1.filters = [myDropShadowFilter];
        s2.filters = [myDropShadowFilter];
     }

     public function createFilters():void {
        // Define filters.
        myBlurFilter = new BlurFilter(4,4,1);

        myGlowFilter = new GlowFilter(color, .8, 6, 6,
            2, 1, false, false);

        myDropShadowFilter = new DropShadowFilter(15, 45,
            color, 0.8, 8, 8, 0.65, 1, false, false);

        myBevelFilter = new BevelFilter(5, 45, color, 0.8,
            0x333333, 0.8, 5, 5, 1, BitmapFilterQuality.HIGH,
            BitmapFilterType.INNER, false);

        applyFilters();
     }
  ]]></mx:Script>
```

```
<mx:Panel>
   <mx:ColumnChart id="myChart" dataProvider="{expenses}">
      <mx:backgroundElements>
         <mx:GridLines id="myGridlines"
              horizontalChangeCount="1"
             verticalChangeCount="1"
             direction="both"
         >
         </mx:GridLines>
      </mx:backgroundElements>
      <mx:horizontalAxis>
         <mx:CategoryAxis
              dataProvider="{expenses}"
              categoryField="Month"
         />
      </mx:horizontalAxis>

      <mx:horizontalAxisRenderer>
         <mx:AxisRenderer id="myAxisRenderer"
              placement="bottom"
             canDropLabels="true"
         >
          <mx:axisStroke>
              <mx:Stroke color="#000080" weight="10"/>
          </mx:axisStroke>
          <mx:tickStroke>
              <mx:Stroke color="#000060" weight="5"/>
          </mx:tickStroke>
          <mx:minorTickStroke>
              <mx:Stroke color="#100040" weight="5"/>
          </mx:minorTickStroke>
         </mx:AxisRenderer>
      </mx:horizontalAxisRenderer>

      <mx:series>
         <mx:ColumnSeries id="s1"
              xField="Month"
              yField="Profit"
              displayName="Profit"
         />
         <mx:ColumnSeries id="s2"
              xField="Month"
              yField="Expenses"
              displayName="Expenses"
         />
      </mx:series>
   </mx:ColumnChart>
   <mx:Legend  id="myLegend" dataProvider="{myChart}"/>
</mx:Panel>
```

```
</mx:Application>
```

Adding grid lines

All charts except the PieChart control have grid lines by default. You can control those grid lines with the CSS `gridLinesStyleName` property, and with the chart series' `backgroundElements` and `annotationElements` properties.

You can include horizontal, vertical, or both grid lines in your chart with the GridLines object. You can set these behind the data series by using the chart's `backgroundElements` property or in front of the data series by using the `annotationElements` property.

The following example turns on grid lines in both directions and applies them to the chart:

```
<?xml version="1.0"?>
<!-- charts/GridLinesBoth.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Array id="bge">
     <mx:GridLines direction="both"/>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        backgroundElements="{bge}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
```

```
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

NOTE The `annotationElements` property refers to any chart elements that appear in the foreground of your chart, and the `backgroundElements` property refers to any chart elements that appear behind the chart's data series.

You can also define the grid lines inside each chart control's definition, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GridLinesBothInternal.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:backgroundElements>
           <mx:GridLines direction="both"/>
        </mx:backgroundElements>

        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
```

```
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

To define the fills and strokes for grid lines, you use the `horizontalStroke`, `verticalStroke`, `horizontalFill`, and `verticalFill` properties. The following properties also define the appearance of grid lines:

- `horizontalAlternateFill`
- `horizontalChangeCount`
- `horizontalOriginCount`
- `horizontalShowOrigin`
- `horiztontalTickAligned`
- `verticalAlternateFill`
- `verticalChangeCount`
- `verticalOriginCount`
- `verticalShowOrigin`
- `verticalTickAligned`

For information on working with strokes, see "Using strokes" on page 1728. For more information on using the `backgroundElements` and `annotationElements` properties, see "Adding ChartElement objects" on page 1690.

You can manipulate the appearance of the grid lines directly in MXML, with ActionScript, or with CSS. The following sections describe techniques for formatting grid lines for Flex Charting objects.

Formatting grid lines with MXML

To control the appearance of the grid lines, you can specify an array of GridLines objects as MXML tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Array id="bge">
     <mx:GridLines
        horizontalChangeCount="1"
        verticalChangeCount="1"
        direction="both"
     >
        <mx:horizontalStroke>
           <mx:Stroke weight="3"/>
        </mx:horizontalStroke>
        <mx:verticalStroke>
           <mx:Stroke weight="3"/>
        </mx:verticalStroke>
        <mx:horizontalFill>
           <mx:SolidColor color="0x99033" alpha=".66"/>
        </mx:horizontalFill>
     </mx:GridLines>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        backgroundElements="{bge}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
```

```
                yField="Profit"
                displayName="Profit"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

This example uses the `changeCount` property to specify that Flex draws grid lines at every tick mark along the axis, and sets the `direction` property to `both`. This causes Flex to draw grid lines both horizontally and vertically. You could also specify `horizontal` or `vertical` as values for the `direction` property.

You can also change the appearance of grid lines by using filters such as a drop shadow, glow, or bevel. For more information, see "Using filters" on page 1747.

Formatting grid lines with CSS

You can set the style of grid lines by applying a CSS style to the GridLines object. The following example applies the `myStyle` style to the grid lines:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Style>
     .myStyle {
        direction:"both";
        horizontalShowOrigin:true;
        horizontalTickAligned:false;
        horizontalChangeCount:1;
        verticalShowOrigin:false;
        verticalTickAligned:true;
        verticalChangeCount:1;
        horizontalFill:#990033;
        horizontalAlternateFill:#00CCFF;
     }
  </mx:Style>

  <mx:Array id="bge">
     <mx:GridLines styleName="myStyle">
        <mx:horizontalStroke>
           <mx:Stroke weight="3"/>
        </mx:horizontalStroke>
        <mx:verticalStroke>
           <mx:Stroke weight="3"/>
        </mx:verticalStroke>
     </mx:GridLines>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        backgroundElements="{bge}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
```

```
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Formatting grid lines with ActionScript

You can manipulate the GridLines at run time with ActionScript. The following example adds filled grid lines in front of and behind the chart's series:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.graphics.SolidColor;
     import mx.graphics.Stroke;
     import mx.charts.GridLines;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     [Bindable]
     public var bge:GridLines;

     public function addGridLines():void {
        bge = new GridLines();

        var s:Stroke = new Stroke(0xff00ff, 2);
        bge.setStyle("horizontalStroke", s);

        var f:SolidColor = new SolidColor(0x990033, .3);
        bge.setStyle("horizontalFill",f);

        var f2:SolidColor = new SolidColor(0x336699, .3);
        bge.setStyle("horizontalAlternateFill",f2);

        myChart.backgroundElements = [bge];
     }
  ]]></mx:Script>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        creationComplete="addGridLines()"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
```

```
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

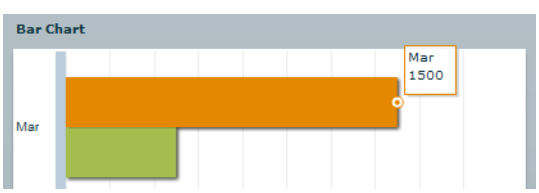
Using DataTips

You use the `showDataTips` property of chart controls to enable DataTip objects. DataTips are similar to Flex ToolTip in that they cause a small pop-up window to appear that shows the data value for the data point under the mouse pointer.

NOTE

DataTip controls and PieChart labels are not the same, although they often show the same information. PieChart labels are always visible regardless of the location of the user's mouse pointer.

The following image shows a simple DataTip:

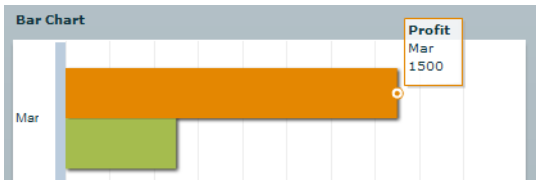


To enable DataTips, set the value of the chart control's `showDataTips` property to `true`, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/EnableDataTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The format of the DataTip depends on the chart type, but typically it displays the names of the fields in the data provider that supply the data at the selected location in the chart and the values of the data from the data provider.

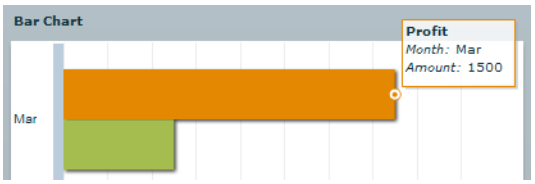
To make the information in the DataTip more understandable to users, you can define the series of your chart with names that are easily understood. Adobe Flash Player displays this name in the DataTip, as the following image shows:



The following example names the data series by using the `displayName` property of the series:

```
<?xml version="1.0"?>
<!-- charts/DataTipsDisplayName.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also name the axis to display labels in the DataTip by sing the `displayName` property. When the axis has a name, this name appears in the DataTip in italic font before the label data, as the following image shows:



In some cases, you add an axis solely for the purpose of adding the label to the DataTip. The following example names both axes so that both data points are labeled in the DataTip:

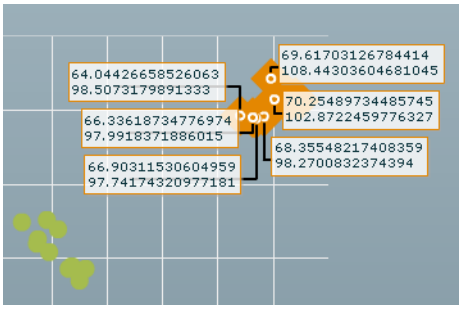
```
<?xml version="1.0"?>
<!-- charts/DataTipsAxisNames.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                displayName="Month"
           />
        </mx:verticalAxis>

        <mx:horizontalAxis>
           <mx:LinearAxis displayName="Amount"/>
        </mx:horizontalAxis>

        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Showing multiple DataTips

You can display more than one DataTip by using the `dataTipMode` property on the chart control. The display options are `single` and `multiple`. When `dataTipMode` is set to `multiple`, the chart displays all DataTips within range of the cursor. The following example sets the value of a ColumnChart control's `dataTipMode` property to `multiple`:

```
<?xml version="1.0"?>
<!-- charts/DataTipsMultiple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
        mouseSensitivity="50"
        dataTipMode="multiple"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The following example shows DataTips when the `dataTipMode` property is set to `multiple`:



The default value of `dataTipMode` depends on the chart type. Its setting is based on the likelihood that there are overlapping DataTips in that chart type. The default value of the `dataTipMode` property for the following chart types is `single`:

- BarChart
- CandlestickChart
- ColumnChart
- HLOCChart
- PieChart

The default value is `multiple` for the `dataTipMode` property for all other chart types.

To determine the size of the interactive area around a data point, you set the `mouseSensitivity` property. The `mouseSensitivity` property configures the distance, in pixels, around the data points where Flex reacts to mouse events such as `click` and `mouseOver`. With this property, you can trigger DataTips to appear when the user moves the mouse pointer *near* the data point rather than *onto* the data point. For more information, see "Changing mouse sensitivity" on page 1815.

Customizing DataTip values

You can customize the text displayed in a DataTip by using the `dataTipFunction` callback function. When you specify a `dataTipFunction` callback function, you can access the data of the DataTip before Flex renders it and customizes the text.

The argument to the callback function is a HitData object. As a result, you must import mx.charts.HitData when using a DataTip callback function.

Flex displays whatever the callback function returns in the DataTip box. You must specify a String as the callback function's return type.

The following example defines a new callback function, `dtFunc`, that returns a formatted value for the DataTip:

```
<?xml version="1.0"?>
<!-- charts/CustomDataTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.charts.HitData;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     public function dtFunc(hd:HitData):String {
        return hd.item.Month + ":<B>$" +
            hd.item.Profit + "</B>";
     }
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
        dataTipFunction="dtFunc"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                displayName="Month"
           />
        </mx:verticalAxis>

        <mx:horizontalAxis>
           <mx:LinearAxis displayName="Amount"/>
        </mx:horizontalAxis>

        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
```

```
            />
        </mx:series>
    </mx:BarChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use the HitData object to get information about the series in which that data item appears. To do this, you cast the HitData object to a Series class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HitDataCasting.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.HitData;
     import mx.charts.series.ColumnSeries;

     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Expenses:1500, Income:1590},
        {Month:"Feb", Expenses:1200, Income:1300},
        {Month:"Mar", Expenses:750, Income:900}
     ]);

     public var b:Boolean = true;

     public function myDataTipFunction(e:HitData):String {
        var s:String;
        s = ColumnSeries(e.element).displayName + "\n";
        s += "Profit: $" + (e.item.Income - e.item.Expenses);
        return s;
     }

  ]]></mx:Script>
  <mx:Panel title="Casting HitData Objects">
     <mx:ColumnChart id="myChart"
        dataProvider="{dataSet}"
        showDataTips="true"
        dataTipFunction="myDataTipFunction"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Expenses"
                displayName="Expenses '06"
           />
```

```
            <mx:ColumnSeries
                 yField="Income"
                 displayName="Income '06"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The series item is also accessible from the HitData object in a custom DataTip function. The `chartItem` property refers to an instance of a subclass of the ChartItem class. The type depends on the series type; for example, the `chartItem` for a ColumnSeries is an instance of the ColumnSeriesItem class.

In the following example, the `yValue` of the ColumnSeriesItem represents the percentage a series takes up in a 100% chart:

```
<?xml version="1.0"?>
<!-- charts/HitDataCastingWithPercent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.HitData;
     import mx.charts.series.ColumnSeries;
     import mx.charts.series.items.ColumnSeriesItem;

     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:1500, Profit:90},
        {Month:"Feb", Income:1200, Profit:100},
        {Month:"Mar", Income:750, Profit:150}
     ]);

     public var b:Boolean = true;

     public function myDataTipFunction(e:HitData):String {

        var s:String;
        s = "<B>" + ColumnSeries(e.element).displayName + "</B>\n";

        s += "<I>Income:</I> <FONT COLOR='#339966'>$" +
            e.item.Income + "</FONT>\n";
        s += "<I>Expenses:</I> <FONT COLOR='#FF0000'>$" +
            (e.item.Income - e.item.Profit) + "</FONT>\n";
        s += "-----------------------\n";
        s += "<I>Profit:</I> $" + e.item.Profit + "\n";

        // The value of the Income will always be 100%,
```

```
        // so exclude adding that to the DataTip. Only
        // add percent when the user gets the Profit DataTip.
        var percentValue:Number =
            Number(ColumnSeriesItem(e.chartItem).yValue);
        if (percentValue < 100) {
            s += "Profit was equal to about <B>" +
                Math.round(percentValue) + "</B>% of the income.\n";
        }
        return s;

        //return e.item.Month + ":<B>$" + e.item.Profit + "</B>";
    }
  ]]></mx:Script>
  <mx:Panel title="Accessing ChartItems from HitData Objects">
     <mx:ColumnChart id="myChart"
        dataProvider="{dataSet}"
        type="100%"
        dataTipFunction="myDataTipFunction"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Profit"
                displayName="Profit '06"
           />
           <mx:ColumnSeries
                yField="Income"
                displayName="Income '06"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

For more information on using the HitData object of chart events, see "Using the HitData object" on page 1804.

The DataTip callback function can return simple HTML for formatted DataTips. Flex supports a small subset of HTML tags including `<FONT>`, `<B>`, `<I>`, and `<BR>`.

Skinning ChartItem objects

A ChartItem object represents a data point in a series. There is one ChartItem instance for each item in the series's data provider. ChartItem objects contain details about the data for the data point as well as the renderer (or, *skin*) to use when rendering that data point in the series. The ChartItem renderers define objects such as the icon that represents a data point in a PlotChart control or the box that makes up a bar in a BarChart control.

Each series has a default renderer that Flex uses to draw that series's ChartItem objects. You can specify a new renderer to use with the series's `itemRenderer` style property. This property points to a class that defines the appearance of the ChartItem object.

You can use existing classes to change the default renderers of chart items. The DiamondItemRenderer class is the default renderer for ChartItem objects in a data series in a PlotChart control. The following example uses the default DiamondItemRenderer class for the first data series. The second series uses the CircleItemRenderer class, which draws a circle to represent the data points in that series. The third series uses the CrossItemRenderer class, which draws a cross shape to represent the data points in that series.

```
<?xml version="1.0"?>
<!-- charts/PlotRenderers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <!-- First series uses default renderer. -->
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
           />

           <!-- Second series uses DiamondItemRenderer. -->
```

```
            <mx:PlotSeries
                 xField="Amount"
                 yField="Expenses"
                 displayName="Plot 2"
                 itemRenderer="mx.charts.renderers.CircleItemRenderer"
            />

            <!-- Third series uses CrossItemRenderer. -->
            <mx:PlotSeries
                 xField="Profit"
                 yField="Amount"
                 displayName="Plot 3"
                 itemRenderer="mx.charts.renderers.CrossItemRenderer"
            />
        </mx:series>
    </mx:PlotChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Using multiple renderer classes

You can sometimes choose more than one renderer for a chart series, depending on the series. These renderers let you easily change the appearance of your charts by adding shadows or graphics.

The following table lists the available renderer classes for the ChartItem objects of each chart type:

Chart type	Available renderer classes
AreaChart	AreaRenderer
BarChart BubbleChart ColumnChart PlotChart	BoxItemRenderer CircleItemRenderer CrossItemRenderer DiamondItemRenderer ShadowBoxItemRenderer TriangleItemRenderer
CandlestickChart HLOCChart	CandlestickItemRenderer HLOCItemRenderer
LineChart	LineRenderer ShadowLineRenderer
PieChart	WedgeItemRenderer

The appearance of most renderers is self-explanatory. The BoxItemRenderer class draws ChartItem objects in the shape of boxes. The DiamondItemRenderer class draws ChartItem objects in the shape of diamonds. The ShadowBoxItemRenderer and ShadowLineRenderer classes add shadows to the ChartItem objects that they draw.

Some series types require multiple renderers to completely render their data. For example, a LineSeries object has both an `itemRenderer` style property and a `lineSegmentRenderer` style property. The `itemRenderer` property specifies the renderer for the data items. The `lineSegmentRenderer` specifies the appearance of the line segments between items.

The other series type that requires two renderers is the AreaSeries. The `areaRenderer` property specifies the appearance of the area, and the `itemRenderer` specifies the appearance of the data items.

You can also specify the renderer to use for the legend marker. The default is the class that the series' `itemRenderer` property specifies. For more information, see "Using Legend controls" on page 1781.

You can use multiple types of data series in a single chart. For example, you can use a ColumnSeries and a LineSeries to show something like a moving average over a stock price. In this case, you can use all the renderers supported by those series in the same chart. For more information on using multiple series, see "Using multiple data series" on page 1671.

Creating custom renderers

You can replace the `itemRenderer` property of a chart series with a custom renderer. You define the renderer on the `itemRenderer` style property for the chart series. This renderer can be a graphical renderer or a class that programmatically defines the renderer.

Creating graphical renderers

You can use a graphic file such as a GIF or JPEG to be used as a renderer on the chart series. You do this by setting the value of the `itemRenderer` style property to be an embedded image. This method of graphically rendering chart items is similar to the graphical skimming method used for other components, as described in "Graphical skinning" on page 807.

The following example uses the graphic file to represent data points on a PlotChart control:

```
<?xml version="1.0"?>
<!-- charts/CustomPlotRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>

           <!-- First series uses embedded image for renderer. -->
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                itemRenderer="@Embed('../assets/bird.gif')"
                radius="50"
                legendMarkerRenderer="@Embed('../assets/bird.gif')"
           />

           <!-- Second series uses DiamondItemRenderer. -->
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.CircleItemRenderer"
           />

           <!-- Third series uses CrossItemRenderer. -->
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"
           />
        </mx:series>
     </mx:PlotChart>
```

```
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

This example uses the bird.gif graphic to represent each data point on the plot chart. It controls the size of the embedded image by using the `radius` property.

You are not required to set the value of the `itemRenderer` property inline. You can also embed a graphic file in ActionScript as a Class, cast it to a ClassFactory, and then reference it inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CustomPlotRendererAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

     [Bindable]
     [Embed(source="../assets/bird.gif")]
     public var myBird:Class;

     [Bindable]
     public var myBirdFactory:ClassFactory =
        new ClassFactory(myBird);

  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>

           <!-- First series uses ActionScript class renderer. -->
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                itemRenderer="{myBirdFactory}"
                legendMarkerRenderer="{myBirdFactory}"
                radius="50"
           />
```

```
            <!-- Second series uses DiamondItemRenderer. -->
            <mx:PlotSeries
                 xField="Amount"
                 yField="Expenses"
                 displayName="Plot 2"
                 itemRenderer="mx.charts.renderers.CircleItemRenderer"
            />

            <!-- Third series uses CrossItemRenderer. -->
            <mx:PlotSeries
                 xField="Profit"
                 yField="Amount"
                 displayName="Plot 3"
                 itemRenderer="mx.charts.renderers.CrossItemRenderer"
            />
        </mx:series>
    </mx:PlotChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Creating programmatic renderers

Creating a custom renderer class for your chart items can give you more control than creating simple graphical renderers. Using class-based renderers is very similar to using programmatic skins, as described in "Programmatic skinning" on page 816.

One approach to is to extend the ProgrammaticSkin class and implement the IDataRenderer interface. In this approach, you can provide all of the logic for drawing chart items in your custom class, and maintain the greatest control over its appearance. For example, you use methods in the Graphics class to draw and fill the rectangles of the bars in a BarChart control.

When you implement the IDataRenderer interface, you must define a setter and getter method to implement the `data` property. This `data` property is of the type of the series item. In the case of a ColumnSeries, it is a ColumnSeriesItem. Other item types include BarSeriesItem, BubbleSeriesItem, LineSeriesItem, and PlotSeriesItem.

In your class, you override the `updateDisplayList()` method with the logic for drawing the chart item as well as setting any custom properties. You should also call the `super.updateDisplayList()` method.

The following example renders the chart items and uses an Array of colors to color each column in the ColumnChart control differently:

```
// charts/CycleColorRenderer.as

package { // Empty package.

  import mx.charts.series.items.ColumnSeriesItem;
  import mx.skins.ProgrammaticSkin;
  import mx.core.IDataRenderer;
  import flash.display.Graphics;

  public class CycleColorRenderer extends mx.skins.ProgrammaticSkin
     implements IDataRenderer {

     private var colors:Array = [0xCCCC99,0x999933,0x999966];
     private var _chartItem:ColumnSeriesItem;

     public function CycleColorRenderer() {
        // Empty constructor.
     }

     public function get data():Object {
        return _chartItem;
     }

     public function set data(value:Object):void {
        _chartItem = value as ColumnSeriesItem;
        invalidateDisplayList();
     }

     override protected function
        updateDisplayList(unscaledWidth:Number,unscaledHeight:Number):void
{
           super.updateDisplayList(unscaledWidth, unscaledHeight);
           var g:Graphics = graphics;
           g.clear();
           g.beginFill(colors[(_chartItem == null)? 0:_chartItem.index]);
           g.drawRect(0, 0, unscaledWidth, unscaledHeight);
           g.endFill();
     }
  } // Close class.
} // Close package.
```

In your Flex application, you use this class as the renderer by using the `itemRenderer` property of the ColumnSeries, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ProgrammaticRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
        [Bindable]
        public var expenses:Object = [
           {Month:"Jan", Profit:2000, Expenses:1500},
           {Month:"Feb", Profit:1000, Expenses:200},
           {Month:"Mar", Profit:1500, Expenses:500}
        ];
    ]]>
  </mx:Script>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:Array>
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
                itemRenderer="CycleColorRenderer"
            />
           </mx:Array>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

For more information on overriding the `updateDisplayList()` method, see "Implementing the updateDisplayList() method" on page 821.

Using Legend controls

Legend controls match the fill patterns on your chart to labels that describe the data series shown with those fill patterns. Each entry in a Legend control is known as a legend item. A legend item contains two basic parts: the marker, and the label. Legend items are of type LegendItem. The following example shows the two parts of a legend item:

In addition to matching fill patterns, legend markers also use the renderer classes of ChartItem objects, such as the points on a PlotChart control. The following example shows a Legend control for a PlotChart control with three series. Each series uses its own renderer class to draw a particular shape, and the Legend control reflects those shapes:

Adding a Legend control to your chart

You use the Legend class to add a legend to your charts. The Legend control displays the label for each data series in the chart and a key that shows the chart element for the series.

The simplest way to create a Legend control is to bind a chart to it by using the `dataProvider` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendNamedSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  </mx:Script>

  <mx:Panel title="Bar Chart with Legend">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

</mx:Application>
```

You add a Legend to a chart in ActionScript by instantiating a new object of type Legend, and then calling the container's `addChild()` method to add the Legend to the container, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="createLegend()">

  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.Legend;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);

     private function createLegend():void {
        var myLegend:Legend = new Legend();
        myLegend.dataProvider = myChart;
        panel1.addChild(myLegend);
     }
  ]]></mx:Script>

  <mx:Panel id="panel1" title="Bar Chart with Legend (Created in
  ActionScript)">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
```

```
    </mx:BarChart>
  </mx:Panel>

</mx:Application>
```

The Legend control creates the legend using information from the chart control. It matches the colors and names of the LegendItem markers to the fills and labels of the chart's data series. In the previous example, Flex uses the BarSeries control's `displayName` property to define the LegendItem label.

Legend controls require that elements of the charts are named. If they are not named, the Legend markers appear, but without labels.

A Legend control for a PieChart control uses the `nameField` property of the data series to find values for the legend. The values that the `nameField` property point to must be Strings.

The following example sets the `nameField` property of a PieChart control's data series to `Expense`. Flex uses the value of the `Expense` field in the data provider in the Legend control.

```
<?xml version="1.0"?>
<!-- charts/LegendNameField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

<mx:Script>
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Expense:"Taxes", Amount:2000},
     {Expense:"Rent", Amount:1000},
     {Expense:"Food", Amount:200}
  ]);
</mx:Script>

  <mx:Panel title="Pie Chart with Legend">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

</mx:Application>
```

The `nameField` property also defines the series for the DataTips and labels.

Creating a custom Legend control

You can create a custom Legend control in MXML by defining the <mx:Legend> tag and populating it with <mx:LegendItem> tags. The following example creates a custom legend for the chart with multiple axes:

```
<?xml version="1.0"?>
<!-- charts/MultipleAxesWithCustomLegend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
      public var SMITH:ArrayCollection = new ArrayCollection([
        {date: "22-Aug-05", close: 41.87},
        {date: "23-Aug-05", close: 45.74},
        {date: "24-Aug-05", close: 42.77},
        {date: "25-Aug-05", close: 48.06},
     ]);

     [Bindable]
      public var DECKER:ArrayCollection = new ArrayCollection([
        {date: "22-Aug-05", close: 157.59},
        {date: "23-Aug-05", close: 160.3},
        {date: "24-Aug-05", close: 150.71},
        {date: "25-Aug-05", close: 156.88},
     ]);

  ]]></mx:Script>

  <mx:Panel title="Column Chart With Multiple Series">
     <mx:ColumnChart id="myChart"
        dataProvider="{SMITH}"
        secondDataProvider="{DECKER}"
        showDataTips="true"
     >

        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{SMITH}"
                categoryField="date"
           />
        </mx:horizontalAxis>

        <mx:verticalAxis>
           <mx:LinearAxis minimum="40" maximum="50"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="cs1"
```

```
                dataProvider="{SMITH}"
                xField="date"
                yField="close"
                displayName="SMITH"
            />
        </mx:series>

        <mx:secondVerticalAxis>
            <mx:LinearAxis minimum="150" maximum="170"/>
        </mx:secondVerticalAxis>

        <mx:secondSeries>
            <mx:LineSeries id="cs2"
                dataProvider="{DECKER}"
                xField="date"
                yField="close"
                displayName="DECKER"
            />
        </mx:secondSeries>
    </mx:ColumnChart>

    <mx:Legend>
        <mx:LegendItem label="SMITH" fontWeight="bold">
            <mx:fill>
             <mx:SolidColor color="0xFF9900"/>
            </mx:fill>
            <mx:stroke>
             <mx:Stroke color="0x000000" weight="1"/>
            </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="DECKER" fontWeight="bold">
            <mx:fill>
             <mx:SolidColor color="{0x999933}"/>
            </mx:fill>
            <mx:stroke>
             <mx:Stroke color="0x000000" weight="1"/>
            </mx:stroke>
         </mx:LegendItem>
    </mx:Legend>
  </mx:Panel>
</mx:Application>
```

Formatting the Legend control

The Legend control is a subclass of the Tile class. You can use Tile properties and some properties of the Container class to format the Legend control. Also, the Legend control has properties (such as `labelPlacement`, `markerHeight`, and `markerWidth`) that you can use to format its appearance. The following table describes the Legend control properties:

Property	Type	Description
`labelPlacement`	String	Specifies the alignment of the LegendItem object's label. Valid values are `right`, `left`, `top`, and `bottom`.
`markerHeight`	Number	Specifies the height, in pixels, of the LegendItem object's marker.
`markerWidth`	Number	Specifies the width, in pixels, of the LegendItem object's marker.
`renderer`	Object	Specifies a class for the LegendItem object's marker. The renderer must implement the IBoxRenderer interface.
`stroke`	Object	Specifies the line stroke for the LegendItem object's marker. For more information on defining line strokes, see "Using strokes" on page 1728.

The following example sets styles by using CSS on the Legend control:

```
<?xml version="1.0"?>
<!-- charts/FormattedLegend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     Legend {
        labelPlacement:left;
        markerHeight:30;
        markerWidth:30;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Bar Chart with Legend">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also change the appearance of the lines on the LegendItem object's marker. You do this with the `stroke` property or the `legendMarkerRenderer` property. For more information, see "Using strokes" on page 1728.

You can place a Legend control anywhere in your application, as long as the control has access to the scope of the chart's data. You can place the Legend control in your application without a container, inside the same container as the chart, or in its own container, such as a Panel container. The latter technique gives the Legend control a border and title bar, and lets you use the `title` attribute of the Panel to create a title, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendInPanel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  </mx:Script>

  <mx:Panel title="Bar Chart with Legend in Panel">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Panel title="Legend">
        <mx:Legend dataProvider="{myChart}"/>
```

```
    </mx:Panel>
  </mx:Panel>

</mx:Application>
```

Setting the direction of legends

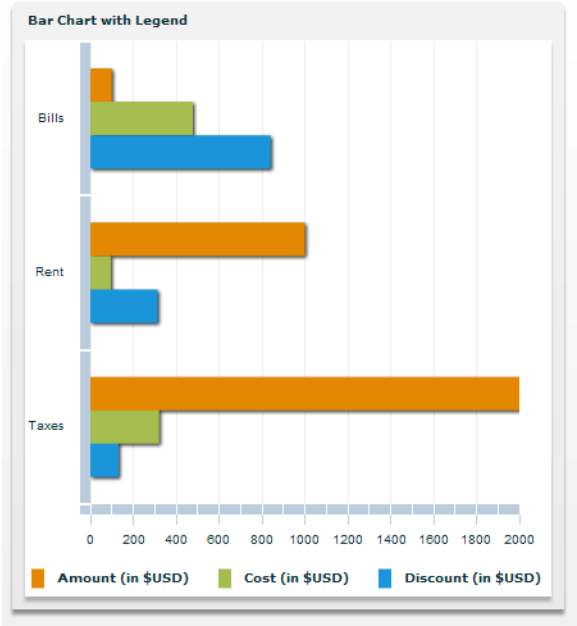
The `direction` property is a commonly used property that is inherited from the Tile container. This property of the `<mx:Legend>` tag causes the LegendItem objects to line up horizontally or vertically. The default value of `direction` is `vertical`; when you use this value, Flex stacks the LegendItem objects one on top of the other.

The following example sets the `direction` property to `horizontal`:

```
<?xml version="1.0"?>
<!-- charts/HorizontalLegend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
backgroundColor="white">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Bar Chart with Legend">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
```

```
    </mx:BarChart>
    <mx:Legend
       dataProvider="{myChart}"
       direction="horizontal"
    />
  </mx:Panel>
</mx:Application>
```

The following example shows the Legend with the `direction` property set to `horizontal`:



Formatting the legend markers

You can define the appearance of the legend markers by using a programmatic renderer class. Flex includes several default renderer classes that you can use for legend markers.

You can change the renderer of the LegendItem object from the default to one of the ChartItem renderers by using the series' `legendMarkerRenderer` style property. This property specifies the class to use when rendering the marker in all associated legends.

The following example sets the legend marker of all three series to the DiamondItemRenderer class:

```
<?xml version="1.0"?>
<!-- charts/CustomLegendRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

     [Bindable]
     [Embed(source="../assets/bird.gif")]
     public var myBird:Class;

     [Bindable]
     public var myBirdFactory:ClassFactory =
        new ClassFactory(myBird);

  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:series>
           <!--
                Each series uses the default renderer for
                the ChartItems, but uses the same renderer
                for legend markers.
           -->

           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"
           />

           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
                legendMarkerRenderer=
```

```
                "mx.charts.renderers.DiamondItemRenderer"
            />

            <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"
            />
        </mx:series>
    </mx:PlotChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If you do not explicitly set the `legendMarkerRenderer` property, the property uses the default class that the series's `itemRenderer` style property specifies. Each series has a default renderer that is used if neither of these style properties is specified.

You can create your own custom legend marker class. Classes used as legend markers must implement the IFlexDisplayObject interface and, optionally, the ISimpleStyleClient and IDataRenderer interfaces.

For more information on available renderer classes, see "Skinning ChartItem objects" on page 1773.

Stacking charts

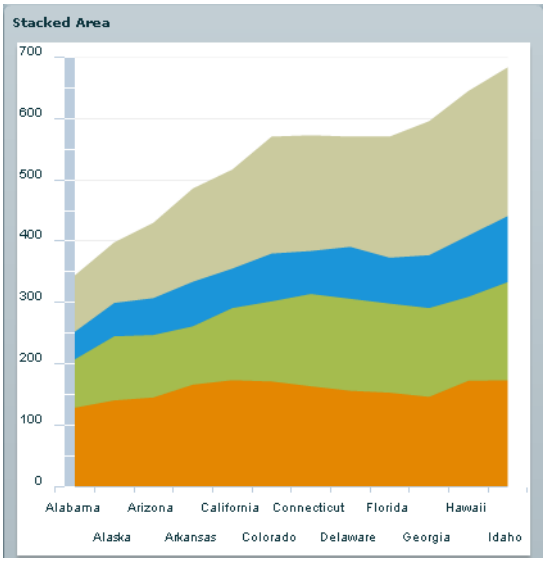
When you chart multiple data series using the AreaChart, BarChart, and ColumnChart controls, you can control how Flex displays the series using the `type` property of the controls. The following table describes the values that the `type` property supports:

Property	Description
`clustered`	Chart elements for each series are grouped by category. This is the default value for BarChart and ColumnChart controls.
`overlaid`	Chart elements for each series are rendered on top of each other, with the element corresponding to the last series on top. This is the default value for AreaChart controls.
`stacked`	Chart elements for each series are stacked on top of each other. Each element represents the cumulative value of the elements beneath it.
`100%`	Chart elements are stacked on top of each other, adding up to 100%. Each chart element represents the percentage that the value contributes to the sum of the values for that category.

The following example creates an AreaChart control that has four data series, stacked on top of each other:

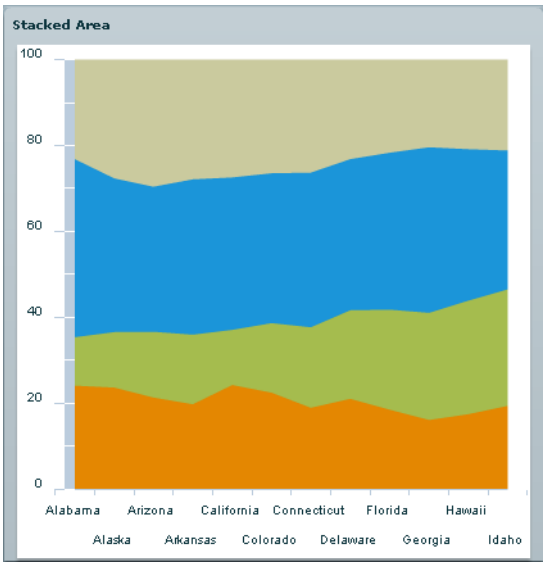
```
<?xml version="1.0"?>
<!-- charts/AreaStacked.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:AreaChart
        dataProvider="{expenses}"
        showDataTips="true"
        type="stacked"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
  </mx:Panel>
</mx:Application>
```

The following example shows a stacked AreaChart:



With an overlay, the last series appears on top, and can obscure the data series below it unless you use the `alpha` property of the fill to make it transparent. For more information, see "Using fills" on page 1735.

If you set the `type` property to 100%, the control draws each series stacked on top of each other, adding up to 100% of the area. Each column represents the percentage that the value contributes to the sum of the values for that category, as the following example shows:



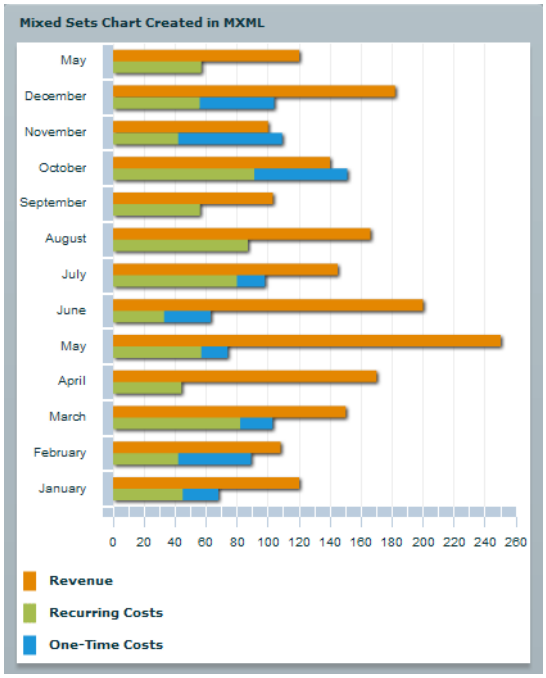
You can use the ColumnSet, BarSet, and AreaSet classes to combine groups of chart series, and thereby use different types of series within the same chart. The following example uses BarSet classes to combine clustered and stacked BarSeries in a single chart. The example shows how to do this in MXML and ActionScript:

```
<?xml version="1.0"?>
<!-- charts/UsingBarSets.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.charts.Legend;
     import mx.charts.BarChart;
     import mx.charts.series.BarSet;
     import mx.charts.series.BarSeries;
     import mx.collections.ArrayCollection;

     [Bindable]
     private var yearlyData:ArrayCollection = new ArrayCollection([
        {month:"January", revenue:120, costs:45,
            overhead:102, oneTime:23},
        {month:"February", revenue:108, costs:42,
            overhead:87, oneTime:47},
        {month:"March", revenue:150, costs:82,
            overhead:32, oneTime:21},
```

```
    {month:"April", revenue:170, costs:44,
        overhead:68},
    {month:"May", revenue:250, costs:57,
        overhead:77, oneTime:17},
    {month:"June", revenue:200, costs:33,
        overhead:51, oneTime:30},
    {month:"July", revenue:145, costs:80,
        overhead:62, oneTime:18},
    {month:"August", revenue:166, costs:87,
        overhead:48},
    {month:"September", revenue:103, costs:56,
        overhead:42},
    {month:"October", revenue:140, costs:91,
        overhead:45, oneTime:60},
    {month:"November", revenue:100, costs:42,
        overhead:33, oneTime:67},
    {month:"December", revenue:182, costs:56,
        overhead:25, oneTime:48},
    {month:"May", revenue:120, costs:57,
        overhead:30}
]);

private function initApp():void {
    var c:BarChart = new BarChart();
    c.dataProvider = yearlyData;

    var vAxis:CategoryAxis = new CategoryAxis();
    vAxis.categoryField = "month";
    c.verticalAxis = vAxis;

    var mySeries:Array = new Array();

    var outerSet:BarSet = new BarSet();
    outerSet.type = "clustered";
    var series1:BarSeries = new BarSeries();
    series1.xField = "revenue";
    series1.displayName = "Revenue";
    outerSet.series = [series1];

    var innerSet:BarSet = new BarSet();
    innerSet.type = "stacked";
    var series2:BarSeries = new BarSeries();
    var series3:BarSeries = new BarSeries();
    series2.xField = "costs";
    series2.displayName = "Recurring Costs";
    series3.xField = "oneTime";
    series3.displayName = "One-Time Costs";
    innerSet.series = [series2, series3];

    c.series = [outerSet, innerSet];
```

```
        var l:Legend = new Legend();
        l.dataProvider = c;

        panel2.addChild(c);
        panel2.addChild(l);
    }
]]></mx:Script>

<mx:Panel title="Mixed Sets Chart Created in MXML" id="panel1">
    <mx:BarChart id="myChart" dataProvider="{yearlyData}">
        <mx:verticalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
            <mx:BarSet type="clustered">
                <mx:BarSeries xField="revenue"
                    displayName="Revenue"/>
                <mx:BarSet type="stacked">
                    <mx:BarSeries
                        xField="costs"
                        displayName="Recurring Costs"/>
                    <mx:BarSeries
                        xField="oneTime"
                        displayName="One-Time Costs"/>
                </mx:BarSet>
            </mx:BarSet>
        </mx:series>
    </mx:BarChart>
    <mx:Legend dataProvider="{myChart}"/>
</mx:Panel>
<mx:Panel title="Mixed Sets Chart Created in ActionScript"
  id="panel2">
</mx:Panel>
</mx:Application>
```

The resulting chart shows two clustered series; one is a standalone series, and the other is a stacked series, as the following example shows:



CHAPTER 56

Using Events and Effects in Charts

56

You can use Adobe Flex Charting to make interesting and engaging charts. Important factors to consider are how users' interactions with your applications trigger effects and events.

Contents

Handling user interactions with charts

Chart controls accept many kinds of user interactions, from moving the mouse over a data point to clicking or double-clicking on that data point. You can create an event handler for each of these interactions and use the Event object in that handler to access the data related to that interaction. For example, if a user clicks on a column in a ColumnChart control, you can access that column's data in the click event handler of that chart.

The chart controls support the mouse events that are inherited from the UIComponent class: `mouseMove`, `mouseOver`, `mouseUp`, `mouseDown`, and `mouseOut`. These events are of type MouseEvent. In addition, the base class for all chart controls, ChartBase, adds several chart data events. The following table describes these events:

Chart data event	Description
`itemClick`	Broadcast when the user clicks the mouse button while over a data point.
`itemDoubleClick`	Broadcast when the user double-clicks the mouse button while over a data point.
`itemMouseDown`	Broadcast when the mouse button is down while over a data point.
`itemMouseMove`	Broadcast when the user moves the mouse pointer while over a data point.
`itemRollOut`	Broadcast when the closest data point under the mouse pointer changes.
`itemRollOver`	Broadcast when the user moves the mouse pointer over a new data point.
`itemMouseUp`	Broadcast when the user releases the mouse button while over a data point.

Chart data events are triggered only when the user moves the mouse pointer over a data point, whereas the UIComponent events are triggered by any mouse interaction with a control.

The chart data events are of type ChartItemEvent. Because ChartItemEvent events are part of the charts package, and not part of the events package, you must import the appropriate classes in the mx.charts.events package to use a ChartItemEvent event.

The following example opens an alert when a the user clicks a data point (a column) in the chart:

```
<?xml version="1.0"?>
<!-- charts/DataPointAlert.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     import mx.charts.events.ChartItemEvent;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Expenses:1500},
        {Month:"Feb", Expenses:200},
        {Month:"Mar", Expenses:500}
     ]);

     public function myHandler(e:ChartItemEvent):void {
        Alert.show("Chart data was clicked");
     }

  ]]></mx:Script>
  <mx:Panel title="Clickable Column Chart">
     <mx:ColumnChart id="myChart"
        itemClick="myHandler(event)"
        dataProvider="{dataSet}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{dataSet}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries yField="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

When you access the chart data events, you can use the HitData object to examine what data was underneath the mouse when the event was triggered. This capability lets you perform actions on specific chart data. For more information, see "Using the HitData object" on page 1804.

Using the HitData object

Flex generates a ChartItemEvent object for each chart data event. In addition to the standard `target` and `type` properties that all Event objects have, Flex adds the `hitData` property to the ChartItemEvent object. This property is an object instance of the HitData class. The `hitData` property contains information about the data point that is closest to the mouse pointer at the time of the mouse event.

The following example uses the itemDoubleClick event handler to display HitData information for data points in a column chart when the user clicks a column. Because each column in the ColumnChart control is associated with a single data point value, clicking the mouse anywhere in the column displays the same information.

```
<?xml version="1.0"?>
<!-- charts/HitDataOnClick.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"init()">
  <mx:Script><![CDATA[
     import mx.charts.events.ChartItemEvent;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Expenses:1500},
        {Month:"Feb", Expenses:200},
        {Month:"Mar", Expenses:500}
     ]);
     // Define the event handler.
     public function myListener(e:ChartItemEvent):void {
        ti1.text = e.hitData.item.Expenses;
        ti2.text = e.hitData.x + "/" + e.hitData.y;
     }

     // Define event listeners when the application initializes.
     public function init():void {
        chart.addEventListener(ChartItemEvent.
            ITEM_DOUBLE_CLICK, myListener);
     }
  ]]></mx:Script>
  <mx:Panel title="Accessing HitData Object in Event Handlers">
     <mx:ColumnChart id="chart"
        dataProvider="{dataSet}" doubleClickEnabled="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries yField="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Form>
        <!--Define a form to display the event information.-->
        <mx:FormItem label="Expenses:">
           <mx:TextInput id="ti1"/>
        </mx:FormItem>
        <mx:FormItem label="x/y:">
           <mx:TextInput id="ti2"/>
        </mx:FormItem>
     </mx:Form>
```

```
  </mx:Panel>
</mx:Application>
```

The `item` property of the HitData object refers to the data point. You can use that property to access its value by its name, as the previous example shows. The HitData *x* and *y* properties refer to the screen coordinates of the data point.

Only data points within the radius determined by the `mouseSensitivity` property can trigger an event on that data point. For more information, see "Changing mouse sensitivity" on page 1815.

Getting chart elements

The HitData object accesses the chart's ChartItem objects. These objects represent data points on the screen. In addition to providing access to the data of data points, ChartItem objects provide information about the size and position of graphical elements that make up the chart. For example, you can get the *x* and *y* positions of columns in a ColumnChart.

The following example uses a semitransparent Canvas container to highlight the data point that the user clicks on with the mouse. The application also accesses the ChartItem object to get the current value to display in a ToolTip on that Canvas:

```
<?xml version="1.0"?>
<!-- charts/ChartItemObjectAccess.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">

  <mx:Style>
     ToolTip {
        fontSize:24;
     }
     ColumnChart {
        gutterLeft: 54;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     import mx.core.IFlexDisplayObject;
     import mx.charts.events.ChartItemEvent;
     import mx.charts.series.items.ColumnSeriesItem;
     import mx.charts.series.ColumnSeries;
     import mx.effects.Move;
     import mx.charts.HitData;
     import mx.collections.ArrayCollection;

     public var m:mx.effects.Move;
     public var hitData:mx.charts.HitData;
     public var chartItem:ColumnSeriesItem;
     public var renderer:IFlexDisplayObject;
     public var series:ColumnSeries;

     public var chartItemPoint:Point;
     public var highlightBoxPoint:Point;
     public var leftAdjust:Number;

     private function init():void {
        m = new Move(highlightBox);

        // This is used to adjust the x location of
        // highlightBox to account for the left
        // gutter width.
        leftAdjust = myChart.getStyle("gutterLeft") - 14;
     }

     [Bindable]
     private var dataSet:ArrayCollection = new ArrayCollection([
        {month:"Jan", income:12300, expense:3210},
        {month:"Feb", income:12450, expense:3100},
```

```
        {month:"Mar", income:13340, expense:3550},
        {month:"Apr", income:13489, expense:3560},
        {month:"May", income:11020, expense:4600},
        {month:"Jun", income:14030, expense:3410},
        {month:"Jul", income:15600, expense:4485},
        {month:"Aug", income:17230, expense:3892},
        {month:"Sep", income:15212, expense:3562},
        {month:"Oct", income:14980, expense:5603},
        {month:"Nov", income:15020, expense:4102},
        {month:"Dec", income:15923, expense:4789}]);

    private function overData(
        event:mx.charts.events.ChartItemEvent):void
    {

        hitData = event.hitData;
        chartItem = ColumnSeriesItem(hitData.chartItem);
        renderer = chartItem.itemRenderer;
        series = ColumnSeries(hitData.element);

        // Add 10 pixels to give it horizontal overlap.
        highlightBox.width = renderer.width * 2 + 10;

        // Add 20 pixels to give it vertical overlap.
        highlightBox.height = renderer.height + 20;

        highlightBoxPoint = new Point(highlightBox.x,
            highlightBox.y);

        // Convert the ChartItem's pixel values from local
        // (relative to the component) to global (relative
        // to the stage). This enables you to place the Canvas
        // container using the x and y values of the stage.
        chartItemPoint = myChart.localToGlobal(new
            Point(chartItem.x, chartItem.y));

        // Define the parameters of the move effect and
        // play the effect.
        m.xTo = chartItemPoint.x + leftAdjust;
        m.yTo = chartItemPoint.y;
        m.duration = 500;
        m.play();

        highlightBox.toolTip = "$" + chartItem.yValue.toString();
    }

]]></mx:Script>

<mx:Panel id="p1">
    <mx:ColumnChart id="myChart"
```

```
        dataProvider="{dataSet}"
        itemClick="overData(event)"
        mouseSensitivity="0"
    >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                displayName="Expense"
                yField="expense"
           />
           <mx:ColumnSeries
                displayName="Income"
                yField="income"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>

  <!-- Define the canvas control that will be used as a highlight -->
  <mx:Canvas id="highlightBox"
        y="0"
        x="0"
        backgroundColor="0xFFFF00"
        alpha=".5"
  />
</mx:Application>
```

For information about changing the appearance of ChartItem objects, see "Skinning ChartItem objects" on page 1773.

Getting data with coordinates

When you create charts, you can use the coordinates on the screen to get the nearest data point or data points, or conversely, pass the data point and get the coordinates.

The `findDataPoints()` and `localToData()` methods take coordinates and return data. The `findDataPoints()` method returns a HitData object. For more information, see "Using the findDataPoints() method" on page 1810. The `localToData()` method returns an Array of the data. For more information, see "Using the localToData() method" on page 1812.

You can also pass the data itself and get the coordinates with the `dataToLocal()` method. For more information, see "Using the dataToLocal() method" on page 1814.

Using the findDataPoints() method

You can use the chart control's `findDataPoints()` method to get an Array of HitData objects by passing in x and y coordinates. If the coordinates do not correspond to the location of a data point, the `findDataPoints()` method returns `null`. Otherwise, the `findDataPoints()` method returns an Array of HitData objects.

The `findDataPoints()` method has the following signature:

```
findDataPoints(x:Number, y:Number):Array
```

The following example creates a PlotChart control and records the location of the mouse pointer as the user moves the mouse over the chart. It uses the `findDataPoints()` method to get an Array of HitData objects, and then displays some of the first object's properties.

```
<?xml version="1.0"?>
<!-- charts/FindDataPoints.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.charts.HitData;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500},
        {Month:"February", Profit:1000, Expenses:200},
        {Month:"March", Profit:1500, Expenses:500},
        {Month:"April", Profit:500, Expenses:300},
        {Month:"May", Profit:1000, Expenses:450},
        {Month:"June", Profit:2000, Expenses:500}]);

     public function handleMouseMove(e:MouseEvent):void {
        // Use coordinates to get HitData object of
        // current data point.
        var hda:Array =
            chart.findDataPoints(e.currentTarget.mouseX,
            e.currentTarget.mouseY);
        if (hda[0]) {
           ta.text = "Found data point " +
                hda[0].chartItem.index + " (x/y):" +
                Math.round(hda[0].x) + "," +
                Math.round(hda[0].y) + "\n";
           ta.text += "Expenses:" + hda[0].item.Expenses;
        } else {
           ta.text = "No data point found  (x/y):" +
                Math.round(e.currentTarget.mouseX) +
                "/" + Math.round(e.currentTarget.mouseY);
        }
     }
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="chart"
        mouseMove="handleMouseMove(event)"
        dataProvider="{expenses}"
        showDataTips="true"
        mouseSensitivity="5"
     >
        <mx:series>
           <mx:PlotSeries
                xField="Profit"
                yField="Expenses"
```

```
            />
        </mx:series>
    </mx:PlotChart>
  </mx:Panel>
  <mx:TextArea id="ta" width="300" height="50"/>
</mx:Application>
```

Using the localToData() method

The `localToData()` method takes a Point object that represents the x and y coordinates you want to get the data for and returns an Array of data values, regardless of whether any data items are at or near that point.

The following example creates a Point object from the mouse pointer's location on the screen and displays the data values associated with that point:

```
<?xml version="1.0"?>
<!-- charts/LocalToData.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     public var p:Point;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     private function updateDetails(e:MouseEvent):void {
        p = new Point(myChart.mouseX,myChart.mouseY);
        mpos.text = "(" + p.x + "," + p.y + ")";

        var d:Array = myChart.localToData(p);
        dval.text = "(" + d[0] + "," + Math.floor(d[1]) + ")";

        p = myChart.dataToLocal(d[0],d[1]);
        dpos.text ="(" + Math.floor(p.x) + "," +
            Math.floor(p.y) + ")";
     }


  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        mouseMove="updateDetails(event)"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
```

```
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
</mx:Panel>


<mx:Form width="300">
    <mx:FormItem label="Mouse Position:">
        <mx:Label id="mpos"/>
    </mx:FormItem>
    <mx:FormItem label="Data Position:">
        <mx:Label id="dpos"/>
    </mx:FormItem>
    <mx:FormItem label="DATA:">
        <mx:Label id="dval"/>
    </mx:FormItem>
</mx:Form>

</mx:Application>
```

Individual chart types determine how coordinates are mapped, and how many values are returned in the Array. The values returned are typically numeric values.

In a chart that is based on the CartesianChart class (for example, a BarChart or ColumnChart control), the first item in the returned Array is the value of the *x*-coordinate along the horizontal axis, and the second item is the value of the *y*-coordinate along the vertical axis.

In a chart based on the PolarChart class (such as PieChart), the returned Array maps the coordinates to a set of polar coordinates—an angle around the center of the chart, and a distance from the center. Those values are mapped to data values that use the first (angular) and second (radial) axes of the chart.

Using the dataToLocal() method

The `dataToLocal()` method converts a set of values to x and y coordinates on the screen. The values you give the method are in the "data space" of the chart; this method converts these values to coordinates. The *data space* is the collection of all possible combinations of data values that a chart can represent.

The number and meaning of arguments passed to the `dataToLocal()` method depend on the chart type. For CartesianChart controls, such as the BarChart and ColumnChart controls, the first value is used to map along the x axis, and the second value is used to map along the y axis.

For PolarChart controls, such as the PieChart control, the first value maps to the angle around the center of the chart, and the second value maps to the distance from the center of the chart along the radius.

The coordinates returned are based on 0,0 being the upper-left corner of the chart. For a ColumnChart control, for example, the height of the column is inversely related to the x coordinate that is returned.

Changing mouse sensitivity

You use the `mouseSensitivity` property of the chart control to determine when the mouse pointer is considered to be over a data point; for example:

```
<mx:ColumnChart id="chart" dataProvider="{dataSet}" mouseSensitivity="30">
```

The current data point is the nearest data point to the mouse pointer that is less than or equal to the number of pixels that the `mouseSensitivity` property specifies.

The default value of the `mouseSensitivity` property is 3 pixels. If the mouse pointer is 4 or more pixels away from a data point, Flex does not trigger a chart data event (such as `itemRollOver` or `itemClick`). Flex still responds to events such as `mouseOver` and `click` by generating an Event object.

The following example initially sets the `mouseSensitivity` property to 20, but lets the user change this value with the HSlider control:

```
<?xml version="1.0"?>
<!-- charts/MouseSensitivity.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

  ]]></mx:Script>
  <mx:Panel title="Mouse Sensitivity">
     <mx:PlotChart id="chart"
        dataProvider="{expenses}"
        showDataTips="true"
        mouseSensitivity="{mySlider.value}"
    >
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="P 1"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="P 2"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="P 3"
           />
        </mx:series>
     </mx:PlotChart>

     <mx:HSlider id="mySlider"
        minimum="0"
        maximum="300"
        value="20"
        dataTipPlacement="top"
        tickColor="black"
```

```
        snapInterval="1"
        tickInterval="20"
        labels="['0','300']"
        allowTrackClick="true"
        liveDragging="true"
    />
  </mx:Panel>

</mx:Application>
```

You can use the `mouseSensitivity` property to increase the area that triggers DataTip events or emits chart-related events. If the mouse pointer is within the range of multiple data points, Flex chooses the closest data point. For DataTip events, if you have multiple DataTip controls enabled, Flex displays all DataTip controls within range. For more information, see "Showing multiple DataTips" on page 1767.

Disabling interactivity

You can make the series in a chart ignore all mouse events by setting the `interactive` property to `false` for that series. The default value is `true`. This lets you disable mouse interactions for one series while allowing it for another.

The following example disables interactivity for events on the first and third PlotSeries objects:

```
<?xml version="1.0"?>
<!-- charts/DisableInteractivity.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

  ]]></mx:Script>
  <mx:Panel title="Disable Interactivity">
     <mx:PlotChart id="chart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="P 1"
                interactive="false"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="P 2"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="P 3"
                interactive="false"
           />
        </mx:series>
     </mx:PlotChart>
  </mx:Panel>
</mx:Application>
```

Disabling the series interactivity has the following results:

- The series does not show DataTip controls.
- The series does not generate a `hitData` structure on any chart data event.

- The series does not return a `hitData` structure when you call the `findDataPoints()` method on the chart.

Using effects with charts

Chart controls support the standard Flex effects such as Zoom and Fade. You can use these effects to make the entire chart control zoom in or fade out in your Flex applications. In addition, chart data series support the following effect classes that apply to the data in the chart:

- SeriesInterpolate
- SeriesSlide
- SeriesZoom

These effects zoom or slide the chart items inside the chart control. These classes are in the mx.charts.effects.effects package. The base class for chart effects is SeriesEffect.

All chart controls and series support the standard Flex triggers and effects that are inherited from UIComponent. These triggers include `focusInEffect`, `focusOutEffect`, `moveEffect`, `showEffect`, and `hideEffect`. The Flex Charting controls also include the `showDataEffect` and `hideDataEffect` triggers.

For information on creating complex effects, see Chapter 17, "Using Behaviors," on page 649.

Using standard effect triggers

Chart controls support standard effects triggers, such as `showEffect` and `hideEffect`.

The following example defines a set of wipe effects that Flex executes when the user toggles the chart's visibility by using the Button control. Also, Flex fades in the chart when it is created.

```
<?xml version="1.0"?>
<!-- charts/StandardEffectTriggers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.effects.Fade;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);

  ]]></mx:Script>

  <!-- Define the effects -->
  <mx:Parallel id="showEffects">
     <mx:WipeRight duration="2000"/>
     <mx:Fade alphaFrom="0" alphaTo="1" duration="4000"/>
  </mx:Parallel>

  <mx:Parallel id="hideEffects">
     <mx:Fade alphaFrom="1" alphaTo="0" duration="2500"/>
     <mx:WipeLeft duration="3000"/>
  </mx:Parallel>

  <mx:Panel title="Area Chart with Effects">
     <mx:AreaChart id="myChart"
        dataProvider="{expenses}"
        creationCompleteEffect="showEffects"
        hideEffect="hideEffects"
        showEffect="showEffects"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
```

```
            />
        </mx:series>
    </mx:AreaChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

  <mx:Button label="Toggle visibility"
        click="myChart.visible=!myChart.visible"
  />
</mx:Application>
```

A negative aspect of using standard effect triggers with charts is that the effects are applied to the entire chart control, and not just the data in the chart control. The result is that an effect such as Fade causes the chart's axes, gridlines, labels, and other chart elements, in addition to the chart's data, to fade in or out during the effect. To solve this, you use special charting effect triggers (see "Using charting effect triggers" on page 1821).

Using charting effect triggers

Charts have two unique effect triggers: `hideDataEffect` and `showDataEffect`. You set these triggers on the data series for the chart. Whenever the data for a chart changes, Flex executes these triggers in succession on the chart's data. The chart control's other elements, such as gridlines, axis lines, and labels are not affected by the effect.

The `hideDataEffect` trigger defines the effect that Flex uses as it hides the current data from view. The `showDataEffect` trigger defines the effect that Flex uses as it moves the current data into its final position on the screen.

Because Flex triggers the effects associated with `hideDataEffect` and `showDataEffect` when the data changes, there is "old" data and "new" data. The effect associated with the `hideDataEffect` trigger is the "old" data that will be replaced by the new data.

The order of events is as follows:

1. Flex first uses the `hideDataEffect` trigger to invoke the effect set for each element of a chart that is about to change. These triggers execute at the same time.
2. Flex then updates the chart with its new data. Any elements (including the grid lines and axes) that do not have effects associated with them are updated immediately with their new values.
3. Flex then invokes the effect set with the `showDataEffect` trigger for each element associated with them. Flex animates the new data in the chart.

Charting effects with data series

Three effects are unique to charting: SeriesInterpolate, SeriesSlide, and SeriesZoom. You use these effects on a data series to achieve an effect when the data in that series changes. These effects can have a great deal of visual impact. Data series do not support other Flex effects.

The following example shows how you can use the SeriesSlide effect to make the data slide in and out of a screen when the data changes. You can trigger changes to data in a chart series in many ways. Most commonly, you trigger an effect when the data provider changes for a chart control. In the following example, when the user clicks the button, the chart controls data provider changes, triggering the effect:

```
<?xml version="1.0"?>
<!-- charts/BasicSeriesSlideEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses1:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:2000, Expenses:1500},
        {Month:"Feb", Income:1000, Expenses:200},
        {Month:"Mar", Income:1500, Expenses:500}
     ]);

     [Bindable]
     public var expenses2:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:1200, Expenses:800},
        {Month:"Feb", Income:2500, Expenses:300},
        {Month:"Mar", Income:575, Expenses:490}
     ]);

     public function changeProvider():void {
        myChart.dataProvider=expenses2;
     }

  ]]></mx:Script>

  <!-- Define chart effects -->
  <mx:SeriesSlide
    id="slideIn"
    duration="1000"
    direction="up"
  />
  <mx:SeriesSlide
    id="slideOut"
    duration="1000"
    direction="down"
  />

  <mx:Panel title="Column Chart with Basic Series Slide Effect">
     <mx:ColumnChart id="myChart" dataProvider="{expenses1}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses1}"
```

```
                categoryField="Month"
            />
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis minimum="0" maximum="3000"/>
        </mx:verticalAxis>

        <mx:series>
            <mx:ColumnSeries
             xField="Month"
             yField="Income"
             displayName="Income"
             showDataEffect="slideIn"
             hideDataEffect="slideOut"
            />
            <mx:ColumnSeries
             xField="Month"
             yField="Expenses"
             displayName="Expenses"
             showDataEffect="slideIn"
             hideDataEffect="slideOut"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
</mx:Panel>

<mx:Button id="b1" click="changeProvider()"
  label="Change Data Provider"
/>

</mx:Application>
```

This example explicitly defines the minimum and maximum values of the vertical axis. If it did not, Flex would recalculate these values when the new data provider was applied. The result would be a change in the axis labels during the effect.

Changing a data provider first triggers the hideDataEffect on the original data provider, which causes that data provider to "slide out," and then triggers the showDataEffect on the new data provider, which causes that data provider to "slide in."

NOTE	If you set the data provider on the series and not the chart control, you must change it on the series and not the chart control.

Another trigger of the data effects is when a new data point is added to a series. The following example triggers the `showDataEffect` when the user clicks the button and adds a new item to the series' data provider:

```
<?xml version="1.0"?>
<!-- charts/AddItemEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var items:ArrayCollection = new ArrayCollection([
            {item: 2000},
            {item: 3300},
            {item: 3000},
            {item: 2100},
            {item: 3200}
        ]);

        public function addDataItem():void {
            // Add a randomly generated value to
            // the data provider.
            var n:Number = Math.random() * 3000;
            var o:Object = {item: n};
            items.addItem(o);
        }
    ]]></mx:Script>

    <!-- Define chart effect -->
    <mx:SeriesSlide id="slideIn"
        duration="1000"
        direction="up"
    />

    <mx:Panel title="Column Chart with Series Effect">
        <mx:ColumnChart id="myChart" dataProvider="{items}">
            <mx:series>
                <mx:ColumnSeries
                    yField="item"
                    displayName="Quantity"
                    showDataEffect="slideIn"
                />
            </mx:series>
        </mx:ColumnChart>
    </mx:Panel>

    <mx:Button id="b1"
        click="addDataItem()"
        label="Add Data Item"
    />
```

```
</mx:Application>
```

For more information about changing charting data at run time, see "Changing chart data at run time" on page 1613.

The charting effects have several properties in common that traditional effects do not have. All of these properties are optional. The following table lists the common properties of the charting effects:

Property	Description
`duration`	The amount of time, in milliseconds, that Flex takes to complete the entire effect. This property defines the speed with which the effect executes. The `duration` property acts as a minimum duration. The effect can take longer based on the settings of other properties. The default value is 500.
`elementOffset`	The amount of time, in milliseconds, that Flex delays the effect on each element in the series. Set the `elementOffset` property to 0 to make all elements start at the same time and end at the same time. Set the `elementOffset` property to an integer such as 30 to stagger the effect on each element by that amount of time. For example, with a slide effect, the first element slides in immediately. The next element begins 30 milliseconds later, and so on. The amount of time for the effect to execute is the same for each element, but the overall duration of the effect is longer. Set the `elementOffset` property to a negative value to make the effect display the last element in the series first. The default value is 20.

Property	Description
`minimumElementDuration`	The amount of time, in milliseconds, that an individual element should take to complete the effect. Charts with a variable number of data points in the series cannot reliably create smooth effects with only the `duration` property. For example, an effect with a duration of 1000 and `elementOffset` of 100 takes 900 milliseconds per element to complete if you have two elements in the series. This is because the start of each effect is offset by 100, and each effect finishes in 1000 milliseconds. If there are four elements in the series, each element takes 700 milliseconds to complete (the last effect starts 300 milliseconds after the first and must be completed within 1000 milliseconds). With 10 elements, each element has only 100 milliseconds to complete the effect. The value of the `minimumElementDuration` property sets a minimal duration for each element. No element of the series takes less than this amount of time (in milliseconds) to execute the effect. As a result, it is possible for an effect to take longer than a specified duration if you specify at least two of the following three properties: `duration`, `offset`, and `minimumElementDuration`. The default value is 0.
`offset`	The amount of time, in milliseconds, that Flex delays the start of the effect. Use this property to stagger effects on multiple series. The default value is 0.

Using the SeriesSlide effect

The SeriesSlide effect slides a data series into and out of the chart's boundaries. The SeriesSlide effect takes a `direction` property that defines the location from which the series slides. Valid values of `direction` are `left`, `right`, `up`, or `down`.

If you use the SeriesSlide effect with a `hideDataEffect` trigger, the series slides from the current position onscreen to a position off the screen, in the direction indicated by the `direction` property. If you use SeriesSlide with a `showDataEffect` trigger, the series slides from off the screen to a position on the screen, in the indicated direction.

When you use the SeriesSlide effect, the entire data series disappears from the chart when the effect begins. The data then reappears based on the nature of the effect. To keep the data on the screen at all times during the effect, you can use the SeriesInterpolate effect. For more information, see "Using the SeriesInterpolate effect" on page 1831.

The following example creates an effect called slideDown. Each element starts its slide 30 milliseconds after the element before it, and takes at least 20 milliseconds to complete its slide. The entire effect takes at least 1 second (1000 milliseconds) to slide the data series down. Flex invokes the effect when it clears old data from the chart and when new data appears.

```
<?xml version="1.0"?>
<!-- charts/CustomSeriesSlideEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item:2000},
        {item:3300},
        {item:3000},
        {item:2100},
        {item:3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item:n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effect -->
  <mx:SeriesSlide duration="1000"
    direction="down"
    minimumElementDuration="20"
    elementOffset="30"
    id="slideDown"
  />

  <mx:Panel title="Column Chart with Custom Series Slide Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="slideDown"
            hideDataEffect="slideDown"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
```

```
  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />
</mx:Application>
```

Using the SeriesZoom effect

The SeriesZoom effect implodes and explodes chart data into and out of the focal point that you specify. As with the SeriesSlide effect, whether the effect is zooming to or from this point depends on whether it's used with a `showDataEffect` or `hideDataEffect` trigger.

The SeriesZoom effect can take several properties that define how the effect acts. The following table describes these properties:

Property	Description
`horizontalFocus` `verticalFocus`	Defines the location of the focal point of the zoom. You combine the `horizontalFocus` and `verticalFocus` properties to define the point from which the data series zooms in and out. For example, set the `horizontalFocus` property to `left` and the `verticalFocus` property to `top` to have the series data zoom to and from the upper-left corner of either element or the chart (depending on the setting of the `relativeTo` property). Valid values of the `horizontalFocus` property are `left`, `center`, `right`, and `undefined`. Valid values of the `verticalFocus` property are `top`, `center`, `bottom`, and `undefined`. If you specify only one of these two properties, the focus is a horizontal or vertical line rather than a point. For example, when you set the `horizontalFocus` property to `left`, the element zooms to and from a vertical line along the left edge of its bounding box. Setting the `verticalFocus` property to `center` causes the element to zoom to and from a horizontal line along the middle of its bounding box. The default value for both properties is `center`.
`relativeTo`	Controls the bounding box used to calculate the focal point of the zooms. Valid values for `relativeTo` are `series` and `chart`. Set the `relativeTo` property to `series` to zoom each element relative to itself. For example, each column of a ColumnChart zooms from the upper-left of the column. Set the `relativeTo` property to `chart` to zoom each element relative to the chart area. For example, each column zooms from the upper-left of the axes, the center of the axes, and so on.

The following example zooms in the data series from the upper-right corner of the chart. While zooming in, Flex displays the last element in the series first because the `elementOffset` value is negative.

```
<?xml version="1.0"?>
<!-- charts/SeriesZoomEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effects -->
  <mx:SeriesZoom id="zoomOut"
     duration="2000"
     minimumElementDuration="50"
     elementOffset="50"
     verticalFocus="top"
     horizontalFocus="left"
     relativeTo="chart"
  />
  <mx:SeriesZoom id="zoomIn"
     duration="2000"
     minimumElementDuration="50"
     elementOffset="-50"
     verticalFocus="top"
     horizontalFocus="right"
     relativeTo="chart"
  />

  <mx:Panel title="Column Chart with Series Zoom Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
```

```
            displayName="Quantity"
            showDataEffect="zoomIn"
            hideDataEffect="zoomOut"
           />
        </mx:series>
    </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</mx:Application>
```

When you use the SeriesZoom effect, the entire data series disappears from the chart when the effect begins. The data then reappears based on the nature of the effect. To have the data stay on the screen at all times during the effect, you can use the SeriesInterpolate effect. For more information, see "Using the SeriesInterpolate effect" on page 1831.

Using the SeriesInterpolate effect

The SeriesInterpolate effect moves the graphics that represent the existing data in the series to the new points. Instead of clearing the chart and then repopulating it as with SeriesZoom and SeriesSlide, this effect keeps the data on the screen at all times.

You use the SeriesInterpolate effect only with the `showDataEffect` effect trigger. It has no effect if set with a `hideDataEffect` trigger.

The following example sets the `elementOffset` property of SeriesInterpolate to 0. As a result, all elements move to their new locations without disappearing from the screen.

```
<?xml version="1.0"?>
<!-- charts/SeriesInterpolateEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effect -->
  <mx:SeriesInterpolate id="rearrangeData"
     duration="1000"
     minimumElementDuration="200"
     elementOffset="0"
  />

  <mx:Panel title="Column Chart with Series Interpolate Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="rearrangeData"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</mx:Application>
```

Applying effects with ActionScript

You can define effects and apply them to chart series by using ActionScript. One way to apply an effect is the same as the way you apply style properties. You use the `setStyle()` method and Cascading Style Sheets (CSS) to apply the effect. For more information on using styles, see Chapter 18, “Using Styles and Themes,” on page 697.

The following example defines the slideIn effect that plays every time the user adds a data item to the chart control's ColumnSeries. The effect is applied to the series by using the `setStyle()` method when the application first loads.

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsAsStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public var rearrangeData:SeriesInterpolate;

     public function init():void {
        rearrangeData = new SeriesInterpolate();
        rearrangeData.duration = 1000;
        rearrangeData.minimumElementDuration = 200;
        rearrangeData.elementOffset = 0;

        // Apply the effect as a style.
        mySeries.setStyle("showDataEffect", "rearrangeData");
     }

     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <mx:Panel title="Column Chart with Series Interpolate Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            id="mySeries"
            yField="item"
            displayName="Quantity"
            showDataEffect="rearrangeData"
           />
```

```
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>

  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />

</mx:Application>
```

When you define an effect in ActionScript, you must ensure that you import the appropriate classes. If you define an effect by using MXML, the compiler imports the class for you.

Instead of applying effects with the `setStyle()` method, you can apply them with CSS if you predefine them in your MXML application; for example:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     ColumnSeries {
        showDataEffect:slideDown;
        hideDataEffect:slideDown;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item:2000},
        {item:3300},
        {item:3000},
        {item:2100},
        {item:3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item:n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effect -->
  <mx:SeriesSlide
    duration="1000"
    direction="down"
    minimumElementDuration="20"
    elementOffset="30"
    id="slideDown"
  />

  <mx:Panel title="Column Chart with Custom Series Slide Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
           />
```

```
        </mx:series>
      </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />
</mx:Application>
```

You can also apply effects with ActionScript and not use styles. You do this by specifying the target of the effect (the series) in the effect's constructor, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public var rearrangeData:SeriesInterpolate;

     public function init():void {
        // Specify the effect's target in the constructor.
        rearrangeData = new SeriesInterpolate(mySeries);
        rearrangeData.duration = 1000;
        rearrangeData.minimumElementDuration = 200;
        rearrangeData.elementOffset = 0;
     }

     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <mx:Panel title="Column Chart with Series Interpolate Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            id="mySeries"
            yField="item"
            displayName="Quantity"
            showDataEffect="rearrangeData"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
```

```
  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />

</mx:Application>
```

You can expand on this example to use slider controls to adjust the effect's properties. To bind the properties of an ActionScript object, such as an effect, to a control, you use methods of the BindingUtils class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithActionScriptSlider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;
     import mx.binding.utils.BindingUtils;

     public var rearrangeData:SeriesInterpolate;
     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public function init():void {
        // Specify the effect's target in the constructor.
        rearrangeData = new SeriesInterpolate(mySeries);

        // Bind effect properties to slider controls
        BindingUtils.bindProperty(rearrangeData,
            "duration", durationSlider, "value");
        BindingUtils.bindProperty(rearrangeData,
            "minimumElementDuration",
            minimumElementDurationSlider, "value");
        BindingUtils.bindProperty(rearrangeData,
            "elementOffset", elementOffsetSlider, "value");
     }

     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }

     public function resetSliders():void {
        durationSlider.value = 1000;
        minimumElementDurationSlider.value = 200;
        elementOffsetSlider.value = 0;
     }
  ]]></mx:Script>
```

```
<mx:Panel title="Column Chart with Series Interpolate Effect">
   <mx:ColumnChart id="myChart" dataProvider="{items}">
      <mx:series>
         <mx:ColumnSeries
          id="mySeries"
          yField="item"
          displayName="Quantity"
          showDataEffect="rearrangeData"
         />
      </mx:series>
   </mx:ColumnChart>
</mx:Panel>

<mx:Button id="b1"
  click="addDataItem()"
  label="Add Data Item"
/>

<mx:Panel>
   <mx:Form>
      <mx:FormItem label="Duration">
         <mx:HSlider id="durationSlider"
          minimum="1"
          maximum="10000"
          value="1000"
          dataTipPlacement="top"
          tickColor="black"
          snapInterval="500"
          tickInterval="500"
          labels="['0','10000']"
          allowTrackClick="true"
          liveDragging="true"
         />
      </mx:FormItem>
      <mx:FormItem label="Minimum Element Duration">
         <mx:HSlider id="minimumElementDurationSlider"
          minimum="0"
          maximum="1000"
          value="200"
          dataTipPlacement="top"
          tickColor="black"
          snapInterval="50"
          tickInterval="50"
          labels="['0','1000']"
          allowTrackClick="true"
          liveDragging="true"
         />
      </mx:FormItem>
      <mx:FormItem label="Element Offset">
```

```
            <mx:HSlider id="elementOffsetSlider"
             minimum="0"
             maximum="1000"
             value="0"
             dataTipPlacement="top"
             tickColor="black"
             snapInterval="50"
             tickInterval="50"
             labels="['0','1000']"
             allowTrackClick="true"
             liveDragging="true"
            />
        </mx:FormItem>
    </mx:Form>
  </mx:Panel>

  <mx:Button id="b2"
    label="Reset Sliders"
    click="resetSliders()"
  />

</mx:Application>
```

For more information about databinding in ActionScript, see "Defining bindings in ActionScript" on page 1260.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z