ADOBE® FLEX® 3
CREATING AND EXTENDING ADOBE FLEX 3 COMPONENTS

Creating and Extending Adobe® Flex® 3 Components

Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA.

Part Number: 90085060 (01/08)

Contents

Part 1: Custom Component Development

Chapter 1: Custom Flex Components

Chapter 2: Custom ActionScript Components

Chapter 3: Custom Events

Chapter 4: Metadata Tags in Custom Components

Chapter 5: Component Compilation

Part 2: MXML Custom Components

Chapter 6: Simple MXML Components

Chapter 7: Advanced MXML Components

Part 3: ActionScript Custom Components

Chapter 8: Simple Visual Components in ActionScript

Chapter 9: Advanced Visual Components in ActionScript

Chapter 10: Custom Style Properties

Chapter 11: Template Components

Part 4: Nonvisual Custom Components

Chapter 12: Custom Formatters

Chapter 13: Custom Validators

Chapter 14: Effects

Part 1: Custom Component Development

Topics

Chapter 1: Custom Flex Components

Adobe® Flex® supports a component-based development model. You use the predefined components included with Flex to build your applications, and create components for your specific application requirements. You can create custom components by using MXML or ActionScript.

Topics

About custom components

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

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

You can build an entire Flex application in a single MXML file that contains both your MXML code and any supporting ActionScript code. As your application gets larger, your single file also grows in size and complexity. This type of application would soon become difficult to understand and debug, and very difficult for multiple developers to work on simultaneously.

Using modules in application development

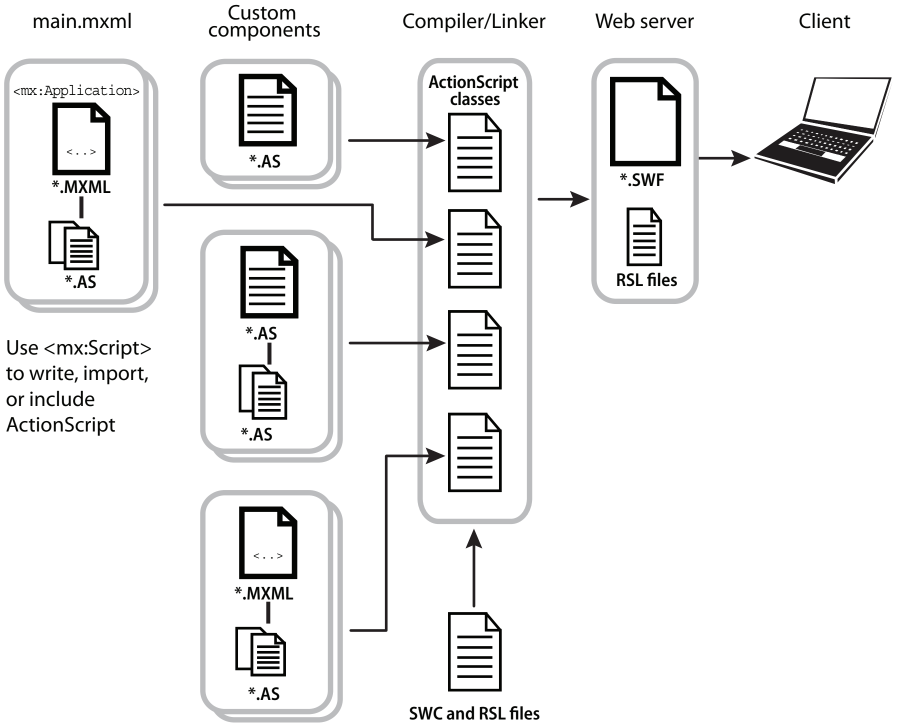
A common coding practice is to divide an application into functional units, or modules, where each module performs a discrete task. Dividing your application into modules provides you with many benefits, including the following:

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

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

Maintainability By developing your application in discrete modules, you can isolate and debug errors faster than you could if you developed your application in a single file.

In Flex, a module corresponds to a custom component, implemented either in MXML or in ActionScript. The following image shows an example of a Flex application divided into components:

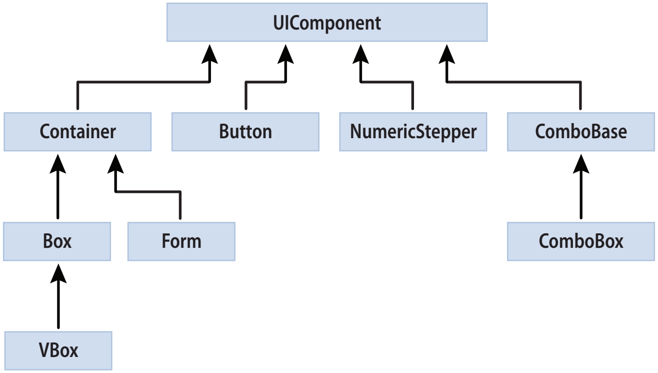


This example shows the following relationships among the components:

- You define a main MXML file that contains the `<mx:Application>` tag.
- In your main MXML file, you define an ActionScript block that uses the `<mx:Script>` tag. Inside the ActionScript block, you write ActionScript code, or include external logic defined by an ActionScript file. Typically, you use this area to write small amounts of ActionScript code. If you must write large amounts of ActionScript code, you should include an external file.
- The main MXML file uses MXML and ActionScript to reference components supplied with Flex, and to reference your custom components.
- Custom components can reference other custom components.

The Flex class hierarchy

Flex is implemented as an ActionScript class hierarchy. That class hierarchy contains component classes, manager classes, data-service classes, and classes for all other Flex features. The following example shows a portion of the class hierarchy for the Flex visual components, such as controls and containers:



***Note:** For a complete description of the class hierarchy, see the Adobe Flex Language Reference.*

All visual components are derived from the UIComponent ActionScript class. Flex nonvisual components are also implemented as a class hierarchy in ActionScript. The most commonly used nonvisual classes are the Validator, Formatter, and Effect base classes.

You create custom components by extending the Flex class hierarchy using the MXML and ActionScript languages. Components inherit the properties, methods, events, styles, and effects of their superclasses.

Customizing existing Flex components

One reason for you to create a component is to customize an existing Flex component for your application requirements. This customization could be as simple as setting the `label` property of a Button control to *Submit* to create a custom button for all of your forms.

You might also want to modify the behavior of a Flex component. For example, a VBox container lays out its children from the top of the container to the bottom in the order in which you define the children within the container. Instead, you might want to customize the VBox container to lay out its children from bottom to top.

Another reason to customize a Flex component is to add logic or behavior to it. For example, you might want to modify the TextInput control so that it supports a key combination to delete all the text entered into the control. Or, you might want to modify a component so that it dispatches a new event type when a user carries out an action.

To create your own components, you create subclasses from the UIComponent class, or any other class in the Flex component hierarchy. For example, if you want to create a component that behaves almost the same as a Button component does, you can extend the Button class instead of recreating all the functionality of the Button class from the base classes.

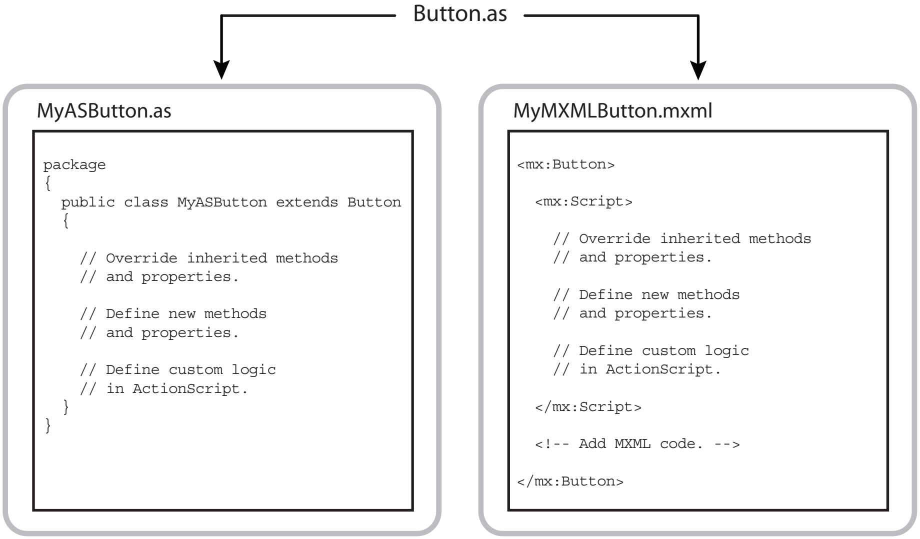
Depending on the modifications that you want to make, you can create a subclass of a Flex component in MXML or ActionScript.

The relationship between MXML components and ActionScript components

To create a custom component in ActionScript, you create a subclass from a class in the Flex class hierarchy. The name of your class (for example, MyASButton), must correspond to the name of the ActionScript file; for example, MyASButton.as. The subclass inherits all of the properties and methods of the superclass. In this example, you use the `<MyASButton>` tag to reference it in MXML.

When you create a custom component in MXML, the Flex compiler automatically creates an ActionScript class. The name of the MXML file (for example, MyMXMLButton.mxml) corresponds to the ActionScript class name. In this example, the ActionScript class is named MyMXMLButton, and you use the `<MyMXMLButton>` tag to reference it in MXML.

The following example shows two components based on the Flex Button component, one defined in ActionScript and the other in MXML:



```
package
{
  public class MyASButton extends Button
  {

    // Override inherited methods
    // and properties.

    // Define new methods
    // and properties.

    // Define custom logic
    // in ActionScript.
  }
}
```

```
<mx:Button>

  <mx:Script>

    // Override inherited methods
    // and properties.

    // Define new methods
    // and properties.

    // Define custom logic
    // in ActionScript.

  </mx:Script>

  <!-- Add MXML code. -->

</mx:Button>
```

Both implementations create a component as a subclass of the Button class and, therefore, inherit all of the public and protected properties, methods, and other elements of the Button class. Within each implementation, you can override inherited items, define new items, and add your custom logic.

***Note:** You cannot override an inherited property defined by a variable, but you can override a property defined by setter and getter methods. You can reset the value of an inherited property defined by a variable. You typically reset it in the constructor of the subclass for an ActionScript component, or in an event handler for an MXML component because MXML components cannot define a constructor.*

However, when you use MXML, the Flex compiler performs most of the overhead required to create a subclass of a component for you. This makes it much easier to create components in MXML than in ActionScript.

Deciding to create components in MXML or ActionScript

One of the first decisions that you must make when creating custom components is deciding whether to write them in MXML or in ActionScript. Ultimately, it is the requirements of your application that determine how you develop your custom component.

Some basic guidelines include the following:

- MXML components and ActionScript components both define new ActionScript classes.
- Almost anything that you can do in a custom ActionScript custom component, you can also do in a custom MXML component. However, for simple components, such as components that modify the behavior of an existing component or add a basic feature to an existing component, it is simpler and faster to create them in MXML.
- When your new component is a composite component that contains other components, and you can express the positions and sizes of those other components using one of the Flex layout containers, you should use MXML to define your component.
- To modify the behavior of the component, such as the way a container lays out its children, use ActionScript.
- To create a visual component by creating a subclass from UIComponent, use ActionScript.
- To create a nonvisual component, such as a formatter, validator, or effect, use ActionScript.
- To add logging support to your control, use ActionScript. For more information, see “Logging” on page 227 in *Building and Deploying Adobe Flex 3 Applications*.

***Note:** The Adobe® Flash® Professional 8 authoring environment does not support ActionScript 3.0. Therefore, you should not use it to create ActionScript components for Flex. Instead, you should use Adobe® Flex® Builder™ or Adobe Flash Professional CS3.*

For more information on custom MXML components, see “Simple MXML Components” on page 63. For more information on ActionScript components, see “Simple Visual Components in ActionScript” on page 105.

Creating new components

Your application might require you to create components, rather than modifying existing ones. To create components, you typically create them in ActionScript by creating a subclass from the UIComponent class. This class contains the generic functionality of all Flex components. You then add the required functionality to your new component to meet your application requirements.

For more information, see “Advanced Visual Components in ActionScript” on page 129.

Creating custom components

You create custom components as either MXML or ActionScript files.

Creating MXML components

Flex supplies a ComboBox control that you can use as part of a form that collects address information from a customer. In the form, you can include a ComboBox control to let the user select the state portion of the address from a list of the 50 states in the U.S. In an application that has multiple forms where a user can enter an address, it would be tedious to create and initialize multiple ComboBox controls with the same information about all 50 states.

Instead, you create an MXML component that contains a ComboBox control with all the 50 states defined within in it. Then, wherever you need to add a state selector to your application, you use your custom MXML component. The following example shows a possible definition for a custom ComboBox control:

```
<?xml version="1.0"?>
<!-- intro\StateComboBox.mxml -->

<!-- Specify the root tag and namespace. -->
<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:dataProvider>
        <mx:String>AK</mx:String>
        <mx:String>AL</mx:String>
        <!-- Add all other states. -->
    </mx:dataProvider>
</mx:ComboBox>
```

This example shows the following:

1 The first line of the custom MXML component definition specifies the declaration of the XML version.

2 The first MXML tag of the component, called its *root* tag, specifies a Flex component or a custom component. MXML components correspond to ActionScript classes, therefore, the root tag specifies the superclass of the MXML component. In this example, the MXML component specifies the Flex ComboBox control as its superclass.

3 The `xmlns` property in the root tag specifies the Flex XML namespace. In this example, the `xmlns` property indicates that tags in the MXML namespace use the prefix *mx:*.

4 The remaining lines of the component specify its definition.

The main application, or any other MXML component file, references the StateComboBox component, as the following example shows:

```
<?xml version="1.0"?>
<!-- intro/IntroMyApplication.mxml -->

<!-- Include the namespace definition for your custom components. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:MyComp="*">

    <!-- Use the filename as the MXML tag name. -->
    <MyComp:StateComboBox/>

</mx:Application>
```

The MXML tag name for a custom component is composed of two parts: the namespace prefix, in this case `MyComp`, and the tag name. The namespace prefix tells Flex where to look for the file that implements the custom component. The tag name corresponds to the filename of the component, in this case StateComboBox.mxml. Therefore, a file named StateComboBox.mxml defines a component with the tag name of `<`*namespace:*`StateComboBox>`.

As part of the `<mx:Application>` tag, the main application file includes the following namespace definition: `xmlns:MyComp="*"`. This definition specifies that the component is in the same directory as the main application file, or in a directory included in the ActionScript source path. For more information on deploying MXML components, see “Simple MXML Components” on page 63.

The best practice is to put your custom components in a subdirectory of your application. That practice helps to ensure that you do not have duplicate component names because they have a different namespace. If you stored your component in the myComponents subdirectory of your application, you would specify the namespace definition as `xmlns:MyComp="myComponents.*"`.

The StateComboBox.mxml file specifies the ComboBox control as its root tag, so you can reference all of the properties of the ComboBox control in the MXML tag of your custom component, or in the ActionScript specified in an `<mx:Script>` tag. For example, the following example specifies the `ComboBox.rowCount` property and a listener for the `ComboBox.close` event for your custom control:

```
<?xml version="1.0"?>
<!-- intro/MyApplicationProperties.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:MyComp="myComponents.*">

    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            private function handleCloseEvent(eventObj:Event):void {
                // ...
            }
        ]]>
    </mx:Script>

    <MyComp:StateComboBox rowCount="5" close="handleCloseEvent(event);"/>

</mx:Application>
```

For more information on MXML components, see "Simple MXML Components" on page 63.

Creating ActionScript components

You create ActionScript components by defining ActionScript classes. You can create the following types of components in ActionScript:

User-interface, or visual, components User-interface components contain both processing logic and visual elements. You create custom user-interface components to modify existing behavior or add new functionality to the component. These components usually extend the Flex component hierarchy. You can extend from the UIComponent class, or any of the Flex components, such as Button, ComboBox, or DataGrid. Your custom ActionScript component inherits all of the methods, properties, events, styles, and effects of its superclass.

Nonvisual components Nonvisual components define nonvisual elements. Flex includes several types of nonvisual components that you can create, including formatters, validators, and effects. You create nonvisual components by creating a subclass from the Flex component hierarchy. For validators, you create subclasses of the Validator class; for formatters you create subclasses of the Formatter class; and for effects, you create subclasses of the Effect class.

For example, you can define a custom button component based on the Flex Button class, as the following example shows:

```
package myComponents
{
    // intro/myComponents/MyButton.as
    import mx.controls.Button;

    public class MyButton extends Button {

        // Define the constructor.
        public function MyButton() {
            // Call the constructor in the superclass.
            super();
            // Set the label property to "Submit".
            label="Submit";
        }
    }
}
```

In this example, you write your MyButton class to the MyButton.as file.

You must define your custom components within an ActionScript package. The package reflects the directory location of your component in the directory structure of your application. Typically, you put custom ActionScript components in directories that are in the ActionScript source path, subdirectories of your application, or for Adobe® LiveCycle™ Data Services ES, in the WEB-INF/flex/user_classes directory. In this example, the `package` statement specifies that the MyButton.as file is in the myComponents subdirectory of your Flex application.

In the MXML file that references the custom component, you define the namespace and reference it in an MXML file as the following example shows:

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

<!-- Include the namespace definition for your custom components. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <!-- Use the filename as the MXML tag name. -->
    <MyComp:MyButton/>

</mx:Application>
```

In this example, you first define the `MyComp` namespace that specifies 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.

For more information, see "Simple Visual Components in ActionScript" on page 105.

Deploying components

When you deploy your custom components as MXML or ActionScript files, you typically deploy them in the same directory structure as your application files, in a directory specified in the ActionScript source path, or for LiveCycle Data Services ES, in the WEB-INF/flex/user_classes directory.

For security reasons, you may decide not to deploy your custom components as source code files. Alternatively, you can deploy your components as SWC files or as part of a Runtime Shared Library (RSL).

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. In addition, the SWF file inside a SWC file is compiled, which means that the code is hidden from casual view.

SWC files can contain one or more components and are packaged and expanded with the PKZip archive format. You can open and examine a SWC file using WinZip, JAR, or another archiving tool. However, you should not manually change the contents of a SWC file, and you should not try to run the SWF file that is in a SWC file outside of a SWC file.

To create a SWC file, use the compc utility in the *flex_install_dir*/bin directory. The compc utility generates a SWC file from MXML component source files and/or ActionScript component source files. For more information on compc, see “Using the Flex Compilers” on page 125 in *Building and Deploying Adobe Flex 3 Applications*.

One way to reduce the size of your application’s SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but only need to be transferred to the client once. These shared files are known as Runtime Shared Libraries or RSLs.

For more information, including information on how to create an RSL file, see “Using Runtime Shared Libraries” on page 195 in *Building and Deploying Adobe Flex 3 Applications*.

For more information on creating different types of components, see the following topics:

- “MXML Custom Components” on page 61
- “ActionScript Custom Components” on page 103
- “Nonvisual Custom Components” on page 181

Chapter 2: Custom ActionScript Components

You use ActionScript code to create ActionScript components for Adobe® Flex®, or to add logic to MXML components. ActionScript provides flow control and object manipulation features that are not available in MXML.

The summary of the general rules for using ActionScript code in custom components in this topic supplements the information in the ActionScript reference documentation. For additional information on ActionScript, see the following resources:

- *Adobe Flex Language Reference*: Contains the API reference for ActionScript 3.0.
- *Programming ActionScript 3.0*: Contains information on using ActionScript 3.0.

Topics

Using ActionScript

Before you start developing custom components, you should be familiar with basic ActionScript coding practices.

Using the package statement

You must define your ActionScript custom components within a package. The package reflects the directory location of your component within the directory structure of your application. To define the package structure, you include the `package` statement in your class definition, as the following example shows:

```
package myComponents
{
    // Class definition goes here.
}
```

Your `package` statement must wrap the entire class definition. If you write your ActionScript class file to the same directory as your other application files, you can leave the package name blank. However, as a best practice, you should store your components in a subdirectory, where the package name reflects the directory location. In this example, write your ActionScript class file to the directory myComponents, a subdirectory of your main application directory.

Formatters are a particular type of component. You might also create a subdirectory of your application's root directory called myFormatters for all of your custom formatter classes. Each formatter class would then define its `package` statement, as the following example shows:

```
package myFormatters
{
    // Formatter class definition goes here.
}
```

If you create a component that is shared among multiple applications, or a component that might be used with third-party components, assign a unique package name to avoid naming conflicts. For example, you might prefix your package name with your company name, as in:

```
package Acme.myFormatters
{
    // Formatter class definition goes here.
}
```

When you reference a custom component from an MXML file, specify a namespace definition for the component that corresponds to its directory location and package name, as the following example shows:

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

    <!-- Declare a formatter and specify formatting properties. -->
    <MyComp:SimpleFormatter id="upperFormat" formatString="upper"/>

    ...

</mx:Application>
```

If a formatter class is in a subdirectory of myFormatters, such as myFormatters/dataFormatters, the package statement is as follows:

```
package myFormatters.dataFormatters
{
    // Formatter class definition goes here.
}
```

You then specify the namespace definition for the component, as the following example shows:

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

    <!-- Declare a formatter and specify formatting properties. -->
    <MyComp:SimpleFormatter id="upperFormat" formatString="upper"/>
    ...

</mx:Application>
```

Using the import statement

You use the `import` statement to import any classes that your class requires. Importing adds a reference to the class so that you can access classes defined by the import. Classes that you import must be located in the ActionScript source path for your application.

You import the classes referenced by your custom component as part of its implementation, as the following example shows:

```
package myComponents
{
    // Import necessary classes.
    import mx.core.Container;
    import mx.controls.Button;
    // Import all classes in the mx.events package
    import mx.events.*;

    // Class definition goes here.

    // You can now create an instance of a Container using this syntax:
    private var myContainer:Container = new Container();

}
```

There is a distinct difference between including and importing in ActionScript. *Including* is copying lines of code from one ActionScript file into another. Files that you include must be located relative to the file performing the include, or use an absolute path. *Importing* is adding a reference to a class file or package so that you can access objects and properties defined by external classes.

For more information on including and importing, see “Using ActionScript” on page 37 in *Adobe Flex 3 Developer Guide*.

Using the class statement

You use the `class` statement to define your class name, and to specify its superclass, as the following example shows:

```
package myComponents
{
    // Import necessary classes
    import mx.core.Container;
    import mx.controls.Button;
    // Import all classes in the mx.events package
    import mx.events.*;

    // Class definition goes here.
    public class MyButton extends Button {

        // Define properties, constructor, and methods.

    }
}
```

The class definition of your component must be prefixed by the `public` keyword, or it cannot be used as an MXML tag. A file that contains a class definition can have one, and only one, public class definition, although it can have additional internal class definitions. Place any internal class definitions at the bottom of your source file below the closing curly brace of the package definition.

In a single ActionScript file, you can define only one class in the package. To define more than one class in a file, define the additional classes outside of the package body.

***Note:** The class definition is one of the few ActionScript constructs that you cannot use in an `<mx:Script>` block in an MXML file.*

Defining the constructor

An ActionScript class must define a public constructor method, which initializes an instance of the class. The constructor has the following characteristics:

- No return type.
- Should be declared public.
- Might have optional arguments.
- Cannot have any required arguments if you use it as an MXML tag.
- Calls the `super()` method to invoke the superclass' constructor.

You call the `super()` method within your constructor to invoke the superclass' constructor to initialize the inherited items from the superclass. The `super()` method should be the first statement in your constructor; otherwise, the inherited parts of the superclass might not be properly constructed. In some cases, you might want to initialize your class first, and then call `super()`.

***Note:** If you do not define a constructor, the compiler inserts one for you and adds a call to `super()`. However, it is considered a best practice to write a constructor and to explicitly call `super()`, unless the class contains nothing but static members. If you define the constructor, but omit the call to `super()`, Flex automatically calls `super()` at the beginning of your constructor.*

In the following example, you define a constructor that uses `super()` to call the superclass' constructor:

```
package myComponents
{
    // Import necessary classes
    import mx.core.Container;
    import mx.controls.Button;
    // Import all classes in the mx.events package
    import mx.events.*;

    // Class definition goes here.
    public class MyButton extends Button {

        // Public constructor.
        public function MyButton()
        {
            // Call the constructor in the superclass.
            super();
        }
        // Define properties and methods.

    }
}
```

***Note:** You cannot define a constructor for an MXML component. For more information, see "About implementing IMXMLObject" on page 100*

Defining properties as variables

Properties let you define data storage within your class. You can define your properties as public, which means that they can be accessed by users of the class. You can also define properties as private, which means that they are used internally by the class, as the following example shows:

```
public class MyButton extends Button {

    // Define private vars.
    private var currentFontSize:Number;

    // Define public vars.
    public var maxFontSize:Number = 15;
    public var minFontSize:Number = 5;
}
```

Users of the class can access the public variables but not the private variables, as the following example shows:

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

    <MyComp:MyButton label="Submit" maxFontSize="30"/>
</mx:Application>
```

Although you can define your classes to use public properties, you may find it advantageous to define properties by using setter and getter methods. For more information, see "Defining methods" on page 19.

***Note:** You cannot override an inherited property defined by a variable, but you can override a property defined by setter and getter methods. You can reset the value of an inherited property defined by a variable. You typically reset it in the constructor of the subclass for an ActionScript component, or in an event handler for an MXML component because MXML components cannot define a constructor.*

Defining properties as getters and setters

You can define properties for your components by using setter and getter methods. The advantage of getters and setters is that they isolate the variable from direct public access so that you can perform the following actions:

- Inspect and validate any data written to the property on a write
- Trigger events that are associated with the property when the property changes
- Calculate a return value on a read
- Allow a child class to override

To define getter and setter methods, precede the method name with the keyword `get` or `set`, followed by a space and the property name. The following example shows the declaration of a public property named `initialCount`, and the getter and setter methods that get and set the value of this property:

```
// Define internal private variable.
private var _initialCount:uint = 42;

// Define public getter.
public function get initialCount():uint {
    return _initialCount;
}

// Define public setter.
public function set initialCount(value:uint):void {
    _initialCount = value;
}
```

By convention, setters use the identifier `value` for the name of the argument.

The variable that stores the property's value cannot have the same name as the getter or setter. By convention, precede the name of the variables with one (_) or two underscores (__). In addition, Adobe recommends that you declare the variable as private or protected.

Users of the class can access the public property, as the following example shows:

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

    <MyComp:MyButton label="Submit" initialCount="24"/>
</mx:Application>
```

If the getter or setter overrides a getter or setter in a superclass, ensure that you include the `override` keyword, as the following example shows:

```
override public function get label():String {}
override public function set label(value:String):void {}
```

Defining methods

Methods define the operations that your class can perform. You define methods in the body of the class. Your methods can override a method of a superclass, or define new functionality for your components.

If the method adds new functionality, you define it using the `function` keyword, as the following example shows:

```
public function myMethod():void {
    // Method definition
}
```

If you define this method as a public method, users of the class can call it.

You can also define private methods, as the following example shows:

```
private function internalMethod():void {
    // Method definition
}
```

Private methods are for internal use by the class, and cannot be called by users of the class.

If the method overrides a method in a superclass, you must include the `override` keyword and the signature of the method must exactly match that of the superclass method, as the following example shows:

```
override protected function createChildren():void {
    // Method definition
}
```

Your methods may take required or optional arguments. To make any of the arguments optional, assign default values to them, as the following example shows:

```
override public validate(value:Object = null,
    supressEvents:Boolean = false):ValidationResultEvent {
    // Method definition
}
```

If the method takes a variable number of arguments, use the "..." syntax, as the following example shows:

```
function foo(n:Number, ... rest):void {
    // Method definition
}
```

Flex creates an Array called `rest` for the optional arguments. Therefore, you can determine the number of arguments passed to the method by using `rest.length`, and access the arguments by using `rest[i]`.

Using the super keyword in a method override

You use the `super` keyword in a method override to invoke the corresponding method of the superclass. The `super` keyword has the following syntax:

```
super.methodName([arg1, ..., argN])
```

This technique is useful when you create a subclass method that adds behavior to a superclass method but also invokes the superclass method to perform its original behavior.

Note: *Although Flex automatically calls the* `super()` *method in a constructor to execute the superclass' constructor, you must call* `super.methodName()` *in a method override. Otherwise, the superclass' version of the method does not execute.*

Whether you call `super.myMethod()` within a method override depends on your application requirement, as follows:

- Typically, you extend the existing functionality of the superclass method, so the most common pattern is to call `super.myMethod()` first in your method override, and then add your logic.
- You might need to change something before the superclass method does its work. In this case, you might call `super.myMethod()` in the override after your logic.
- In some method overrides, you might not want to invoke the superclass method at all. Only call `super.myMethod()` if and when you want the superclass to do its work.
- Sometimes the superclass has an empty method that does nothing, which requires you to implement the functionality in the method. In this case, you should still call `super.myMethod()` because in a future version of Flex, that method might implement some functionality. For more information, see the documentation on each Flex class.

About the scope

Scoping is mostly a description of what the `this` keyword refers to at any given point in your application. In the main MXML application file, the file that contains the `<mx:Application>` tag, the current scope is the Application object, and therefore the `this` keyword refers to the Application object.

In an ActionScript component, the scope is the component itself and not the application or other file that references the component. As a result, the `this` keyword inside the component refers to the component instance and not the Flex Application object.

Nonvisual ActionScript components do not have access to their parent application with the `parentDocument` property. However, you can access the top-level Application object by using the `mx.core.Application.application` property.

For more information on scope, see “Using ActionScript” on page 37 in *Adobe Flex 3 Developer Guide*.

Chapter 3: Custom Events

You can create custom events as part of defining MXML and ActionScript components. Custom events let you add functionality to your custom components to respond to user interactions, to trigger actions by your custom component, and to take advantage of data binding.

For more information on creating custom events for MXML components, see "Advanced MXML Components" on page 79. For information on creating custom events for ActionScript components, see "Simple Visual Components in ActionScript" on page 105.

Topics

About events

Adobe Flex applications are event-driven. Events let an application know when the user interacts with the interface, and also when important changes happen in the appearance or life cycle of a component, such as the creation of a component or its resizing. Events can be generated by user input devices, such as the mouse and keyboard, or by the asynchronous operations, such as the return of a web service call or the firing of a timer.

The core class of the Flex component architecture, mx.core.UIComponent, defines core events, such as `updateComplete`, `resize`, `move`, `creationComplete`, and others that are fundamental to all components. Subclasses of UIComponent inherit these events.

Custom components that extend existing Flex classes inherit all the events of the base class. Therefore, if you extend the Button class to create the MyButton class, you can use the `click` event and the events that all controls inherit, such as `mouseOver` or `initialize`, as the following example shows.

```
<?xml version="1.0"?>
<!-- events/EventsMyApplication.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            // Event listener for the click event.
            private function handleClick(eventObj:Event):void {
                // Define event listener.
            }

            // Event listener for the initialize event.
            private function handleInit(eventObj:Event):void {
                // Define event listener.
            }
        ]]>
    </mx:Script>

    <MyComp:MyButton
        click="handleClick(event);"
        initialize="handleInit(event);"/>

</mx:Application>
```

In addition to using the events inherited from its superclasses, your custom components can define custom events. You use custom events to support data binding, to respond to user interactions, or to trigger actions by your component.

For more information on the Flex event mechanism, see “Using Events” on page 61 in *Adobe Flex 3 Developer Guide*.

Using an event object

When a Flex component dispatches an event, it creates an event object, where the properties of the event object contain information describing the event. An event listener takes this event object as an argument and accesses the properties of the object to determine information about the event.

The base class for all event objects is the flash.events.Event class. All event objects are instances of the Event class, or instances of a subclass of the Event class.

The following table describes the public properties of the Event class. The Event class implements these properties using getter methods.

Property	**Type**	**Description**
`type`	String	The name of the event; for example, `"click"`. The event constructor sets this property.
`target`	EventDispatcher	A reference to the component instance that dispatches the event. This property is set by the `dispatchEvent()` method; you cannot change this to a different object.
`currentTarget`	EventDispatcher	A reference to the component instance that is actively processing the Event object. The value of this property is different from the value of the `target` property during the event capture and bubbling phase. For more information, see "Using Events" on page 61 in *Adobe Flex 3 Developer Guide*.
`eventPhase`	uint	The current phase in the event flow. The property might contain the following values: • `EventPhase.CAPTURING_PHASE`: The capture phase • `EventPhase.AT_TARGET`: The target phase • `EventPhase.BUBBLING_PHASE`: The bubbling phase
`bubbles`	Boolean	Whether an event is a bubbling event. If the event can bubble, the value for this property is `true`; otherwise, it is `false`. You can optionally pass this property as a constructor argument to the Event class. By default, most event classes set this property to `false`. For more information, see "Using Events" on page 61 in *Adobe Flex 3 Developer Guide*.
`cancelable`	Boolean	Whether the event can be canceled. If the event can be canceled, the value for this value is `true`; otherwise, it is `false`. You can optionally pass this property as a constructor argument to the Event class. By default, most event classes set this property to `false`. For more information, see "Using Events" on page 61 in *Adobe Flex 3 Developer Guide*.

Dispatching custom events

Flex defines many of the most common events, such as the `click` event for the Button control; however, your application may require that you create events. In your custom Flex components, you can dispatch any of the predefined events inherited by the component from its superclass, and dispatch new events that you define within the component.

To dispatch a new event from your custom component, you must do the following:

1 (Optional) Create a subclass from the flash.events.Event class to create an event class that describes the event object. For more information, see "Creating a subclass from the Event class" on page 26.

2 (Optional) Use the `[Event]` metadata tag to make the event public so that the MXML compiler recognizes it. For more information, see "Using the Event metadata tag" on page 27.

3 Dispatch the event using the `dispatchEvent()` method. For more information, see "Dispatching an event" on page 29.

Creating a subclass from the Event class

All events use an event object to transmit information about the event to the event listener, where the base class for all event objects is the flash.events.Event class. When you define a custom event, you can dispatch an event object of the Event type, or you can create a subclass of the Event class to dispatch an event object of a different type. You typically create a subclass of the Event class when your event requires you to add information to the event object, such as a new property to hold information that the event listener requires.

For example, the event objects associated with the Flex Tree control include a property named `node`, which identifies the node of the Tree control associated with the event. To support the `node` property, the Tree control dispatches event objects of type TreeEvent, a subclass of the Event class.

Within your subclass of the Event class, you can add properties, add methods, set the value of an inherited property, or override methods inherited from the Event class. For example, you might want to set the `bubbles` property to `true` to override the default setting of `false`, which is inherited from the Event class.

You are required to override the `Event.clone()` method in your subclass. The `clone()` method returns a cloned copy of the event object by setting the `type` property and any new properties in the clone. Typically, you define the `clone()` method to return an event instance created with the `new` operator.

Suppose that you want to pass information about the state of your component to the event listener as part of the event object. To do so, you create a subclass of the Event class to create an event, EnableChangeEvent, as the following example shows:

```
package myEvents
{
    //events/myEvents/EnableChangeEvent.as
    import flash.events.Event;

    public class EnableChangeEvent extends Event
    {

        // Public constructor.
        public function EnableChangeEvent(type:String,
            isEnabled:Boolean=false) {
                // Call the constructor of the superclass.
                super(type);

                // Set the new property.
                this.isEnabled = isEnabled;
        }

        // Define static constant.
        public static const ENABLE_CHANGED:String = "enableChanged";

        // Define a public variable to hold the state of the enable property.
        public var isEnabled:Boolean;

        // Override the inherited clone() method.
        override public function clone():Event {
            return new EnableChangeEvent(type, isEnabled);
        }
    }
}
```

In this example, your custom class defines a public constructor that takes two arguments:

- A String value that contains the value of the `type` property of the Event object.
- An optional Boolean value that contains the state of the component's `isEnabled` property. By convention, all constructor arguments for class properties, except for the `type` argument, are optional.

From within the body of your constructor, you call the `super()` method to initialize the base class properties.

Using the Event metadata tag

You use the `[Event]` metadata tag to define events dispatched by a component so that the Flex compiler can recognize them as MXML tag attributes in an MXML file. You add the `[Event]` metadata tag in one of the following locations:

ActionScript components Above the class definition, but within the package definition, so that the events are bound to the class and not a particular member of the class.

MXML components In the `<mx:Metadata>` tag of an MXML file.

The `Event` metadata keyword has the following syntax:

```
[Event(name=”eventName”, type=”package.eventType”)]
```

The *`eventName`* argument specifies the name, including the package, of the event. The *`eventType`* argument specifies the class that defines the event.

The following example identifies the `enableChange` event as an event that an ActionScript component can dispatch:

```
[Event(name="enableChange", type="myEvents.EnableChangeEvent")]
public class MyComponent extends TextArea
{
    ...
}
```

The following example shows the `[Event]` metadata tag within the `<mx:Metadata>` tag of an MXML file:

```
<?xml version="1.0"?>
<!-- events\myComponents\MyButton.mxml -->

<mx:Button xmlns:mx="http://www.adobe.com/2006/mxml"
    click="dispatchEvent(new EnableChangeEvent('enableChanged'));">

    <mx:Script>
        <![CDATA[
            import myEvents.EnableChangeEvent;
        ]]>
    </mx:Script>

    <mx:Metadata>
        [Event(name="enableChanged", type="myEvents.EnableChangeEvent")]
    </mx:Metadata>

</mx:Button>
```

Once you define the event using the `[Event]` metadata tag, you can refer to the event in an MXML file, as the following example shows:

```
<?xml version="1.0"?>
<!-- events/MainEventApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <mx:Script>
        <![CDATA[
            import myEvents.EnableChangeEvent;

            public function
                enableChangedListener(eventObj:EnableChangeEvent):void {
                    // Handle event.
            }
        ]]>
    </mx:Script>
    <MyComp:MyButton enableChanged="myTA.text='got event';" />

    <mx:TextArea id="myTA" />

</mx:Application>
```

If you do not identify an event with the `[Event]` metadata tag, the compiler generates an error if you try to use the event name in MXML. The metadata for events is inherited from the superclass, however, so you do not need to tag events that are already defined with the `[Event]` metadata tag in the superclass.

Dispatching an event

You use the `dispatchEvent()` method to dispatch an event. The `dispatchEvent()` method has the following signature:

```
public dispatchEvent(event:Event):Boolean
```

This method requires an argument of the Event type, which is the event object. The `dispatchEvent()` method initializes the `target` property of the event object with a reference to the component dispatching the event.

You can create an event object and dispatch the event in a single statement, as the following example shows:

```
dispatchEvent(new Event("click"));
```

You can also create an event object, initialize it, and then dispatch it, as the following example shows:

```
var eventObj:EnableChangeEvent = new EnableChangeEvent("enableChange");
eventObj.isEnabled=true;
dispatchEvent(eventObj);
```

For complete examples that create and dispatch custom events, see “Advanced MXML Components” on page 79 and “Simple Visual Components in ActionScript” on page 105.

Creating static constants for the Event.type property

The constructor of an event class typically takes a single required argument that specifies the value of the event object’s `type` property. In the previous section, you passed the string `enableChange` to the constructor, as the following example shows:

```
// Define event object, initialize it, then dispatch it.
var eventObj:EnableChangeEvent = new EnableChangeEvent("enableChange");
dispatchEvent(eventObj);
```

The Flex compiler does not examine the string passed to the constructor to determine if it is valid. Therefore, the following code compiles, even though `enableChangeAgain` might not be a valid value for the `type` property:

```
var eventObj:EnableChangeEvent =
    new EnableChangeEvent("enableChangeAgain");
```

Because the compiler does not check the value of the `type` property, the only time that your application can determine if `enableChangeAgain` is valid is at run time.

However, to ensure that the value of the `type` property is valid at compile time, Flex event classes define static constants for the possible values for the `type` property. For example, the Flex EffectEvent class defines the following static constant:

```
// Define static constant for event type.
public static const EFFECT_END:String = "effectEnd";
```

To create an instance of an EffectEvent class, you use the following constructor:

```
var eventObj:EffectEvent = new EffectEvent(EffectEvent.EFFECT_END);
```

If you incorrectly reference the constant in the constructor, the compiler generates a syntax error because it cannot locate the associated constant. For example, the following constructor generates a syntax error at compile time because `MY_EFFECT_END` is not a predefined constant of the EffectEvent class:

```
var eventObj:EffectEvent = new EffectEvent(EffectEvent.MY_EFFECT_END);
```

You can use this technique when you define your event classes. The following example modifies the definition of the EnableChangeEventConst class to include a static constant for the `type` property:

```
package myEvents
{
    //events/myEvents/EnableChangeEventConst.as
    import flash.events.Event;

    public class EnableChangeEventConst extends Event
    {
        // Public constructor.
        public function EnableChangeEventConst(type:String,
            isEnabled:Boolean=false) {
                // Call the constructor of the superclass.
                super(type);

                // Set the new property.
                this.isEnabled = isEnabled;
        }

        // Define static constant.
        public static const ENABLE_CHANGED:String = "myEnable";

        // Define a public variable to hold the state of the enable property.
        public var isEnabled:Boolean;

        // Override the inherited clone() method.
        override public function clone():Event {
            return new EnableChangeEvent(type, isEnabled);
        }
    }
}
```

Now you create an instance of the class by using the static constant, as the following example shows for the MyButtonConst custom component:

```
<?xml version="1.0"?>
<!-- events\myComponents\MyButtonConst.mxml -->

<mx:Button xmlns:mx="http://www.adobe.com/2006/mxml"
    click="dispatchEvent(new
EnableChangeEventConst(EnableChangeEventConst.ENABLE_CHANGED));">

    <mx:Script>
        <![CDATA[
            import myEvents.EnableChangeEventConst;
        ]]>
    </mx:Script>

    <mx:Metadata>
        [Event(name="myEnable", type="myEvents.EnableChangeEventConst")]
    </mx:Metadata>

</mx:Button>
```

This technique does not preclude you from passing a string to the constructor.

Chapter 4: Metadata Tags in Custom Components

You insert metadata tags into your MXML and ActionScript files to provide information to the Adobe® Flex® compiler. Metadata tags do not get compiled into executable code, but provide information to control how portions of your code get compiled.

For more information about additional metadata tags that you use when creating an application, such as the `[Embed]` metadata tag, see “Embedding Assets” on page 965 in *Adobe Flex 3 Developer Guide*.

Topics

About metadata tags

Metadata tags provide information to the Flex compiler that describes how your components are used in a Flex application. For example, you might create a component that defines a new event. To make that event known to the Flex compiler so that you can reference it in MXML, you insert the `[Event]` metadata tag into your component, as the following ActionScript class definition shows:

```
[Event(name="enableChanged", type="flash.events.Event")]
class ModalText extends TextArea {
    ...
}
```

In this example, the `[Event]` metadata tag specifies the event name and the class that defines the type of the event object dispatched by the event. After you identify the event to the Flex compiler, you can reference it in MXML, as the following example shows:

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

    <mx:Script>
        <![CDATA[
            function handleEnableChangeEvent(eventObj:Event):void {
                ...
            }
        ]]>
    </mx:Script>

    <MyComp:ModalText enableChanged="handleEnableChangeEvent(event);"/>

</mx:Application>
```

If you omit the `[Event]` metadata tag from your class definition, Flex issues a syntax error when it compiles your MXML file. The error message indicates that Flex does not recognize the `enableChanged` property.

The Flex compiler recognizes component metadata statements in your ActionScript class files and MXML files. The metadata tags define component attributes, data binding properties, events, and other properties of the component. Flex interprets these statements during compilation; they are never interpreted during run time.

Metadata statements are associated with a class declaration, an individual data field, or a method. They are bound to the next line in the file. When you define a component property or method, add the metadata tag on the line before the property or method declaration.

Metadata tags in ActionScript

In an ActionScript file, when you define component events or other aspects of a component that affect more than a single property, you add the metadata tag outside the class definition so that the metadata is bound to the entire class, as the following example shows:

```
// Add the [Event] metadata tag outside of the class file.
[Event(name="enableChange", type="flash.events.Event")]
public class ModalText extends TextArea {

    ...

    // Define class properties/methods
    private var _enableTA:Boolean;

    // Add the [Inspectable] metadata tag before the individual property.
    [Inspectable(defaultValue="false")]
    public function set enableTA(val:Boolean):void {
        _enableTA = val;
        this.enabled = val;

        // Define event object, initialize it, then dispatch it.
        var eventObj:Event = new Event("enableChange");
        dispatchEvent(eventObj);
    }
}
```

In this example, you add the `[Event]` metadata tag before the class definition to indicate that the class dispatches an event named `enableChanged`. You also include the `[Inspectable]` metadata tag to indicate the default value of the property for Adobe® Flex® Builder™. For more information on using this tag, see “Inspectable metadata tag” on page 44.

Metatdata tags in MXML

In an MXML file, you insert the metadata tags either in an `<mx:Script>` block along with your ActionScript code, or in an `<mx:Metadata>` block, as the following example shows:

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

    <mx:Metadata>
        [Event(name="enableChange", type="flash.events.Event")]
    </mx:Metadata>

    <mx:Script>
        <![CDATA[

            // Import Event class.
            import flash.events.Event;

            // Define class properties and methods.
            private var _enableTA:Boolean;

            // Add the [Inspectable] metadata tag before the individual property.
            [Inspectable(defaultValue="false")]
            public function set enableTA(val:Boolean):void {
                _enableTA = val;
                this.enabled = val;

                // Define event object, initialize it, then dispatch it.
                var eventObj:Event = new Event("enableChange");
                dispatchEvent(eventObj);
            }
        ]]>
    </mx:Script>
</mx:TextArea>
```

A key difference between the `<mx:Metadata>` and `<mx:Script>` tags is that text within the `<mx:Metadata>` tag is inserted before the generated class declaration, but text within `<mx:Script>` tag is inserted in the body of the generated class declaration. Therefore, metadata tags like `[Event]` and `[Effect]` must go in an `<mx:Metadata>` tag, but the `[Bindable]` and `[Embed]` metadata tags must go in an `<mx:Script>` tag.

Metadata tags

The following table describes the metadata tags that you can use in ActionScript class files:

Tag	**Description**
`[ArrayElementType]`	Defines the allowed data type of each element of an Array. For more information, see "ArrayElementType metadata tag" on page 37.
`[Bindable]`	Identifies a property that you can use as the source of a data binding expression. For more information, see "Bindable metadata tag" on page 38.
`[DefaultProperty]`	Defines the name of the default property of the component when you use the component in an MXML file. For more information, see "DefaultProperty metadata tag" on page 40.
`[Deprecated]`	Marks a class or class element as deprecated so that the compiler can recognize it and issue a warning when the element is used in an application. For more information, see "Deprecated metadata tag" on page 41.
`[Effect]`	Defines the MXML property name for the effect. For more information, see "Effect metadata tag" on page 41.
`[Embed]`	Imports JPEG, GIF, PNG, SVG, and SWF files at compile time. Also imports image assets from SWC files. This is functionally equivalent to the MXML `@Embed` syntax, as described in "Embedding Assets" on page 965 in *Adobe Flex 3 Developer Guide*.
`[Event]`	Defines the MXML property for an event and the data type of the event object that a component emits. For more information, see "Event metadata tag" on page 42.
[Exclude]	Omits the class element from the Flex Builder tag inspector. The syntax is as follows: `[Exclude(name="label", kind="property")]`
[ExcludeClass]	Omits the class from the Flex Builder tag inspector. This is equivalent to the `@private` tag in ASDoc when applied to a class.
`[IconFile]`	Identifies the filename for the icon that represents the component in the Insert bar of Adobe Flex Builder. For more information, see "IconFile metadata tag" on page 43.
`[Inspectable]`	Defines an attribute exposed to component users in the attribute hints and Tag inspector of Flex Builder. Also limits allowable values of the property. For more information, see "Inspectable metadata tag" on page 44.
`[InstanceType]`	Specifies the allowed data type of a property of type IDeferredInstance. For more information, see "InstanceType metadata tag" on page 46.

Tag	Description
`[NonCommittingChangeEvent]`	Identifies an event as an interim trigger. For more information, see "NonCommittingChangeEvent metadata tag" on page 47.
[RemoteClass]	Maps the ActionScript object to a Java object. For more information on using the `[RemoteClass]` metadata tag, see "RemoteClass metadata tag" on page 47.
`[Style]`	Defines the MXML property for a style property for the component. For more information on using the `[Style]` metadata tag, see "Style metadata tag" on page 48.
`[Transient]`	Identifies a property that should be omitted from data that is sent to the server when an ActionScript object is mapped to a Java object using `[RemoteClass]`. For more information, see "Transient metadata tag" on page 49.

ArrayElementType metadata tag

When you define an Array variable in ActionScript, you specify `Array` as the data type of the variable. However, you cannot specify the data type of the elements of the Array.

To allow the Flex MXML compiler to perform type checking on Array elements, you can use the `[ArrayElementType]` metadata tag to specify the allowed data type of the Array elements, as the following example shows:

```
public class MyTypedArrayComponent extends VBox {

    [ArrayElementType("String")]
    public var newStringProperty:Array;

    [ArrayElementType("Number")]
    public var newNumberProperty:Array;
    ...
}
```

Note: *The MXML compiler checks for proper usage of the Array only in MXML code; it does not check Array usage in ActionScript code.*

In this example, you specify String as the allowed data type of the Array elements. If a user attempts to assign elements of a data type other than String to the Array in an MXML file, the compiler issues a syntax error, as the following example shows:

```
<MyComp:MyTypedArrayComponent>
    <MyComp:newStringProperty>
        <mx:Number>94062</mx:Number>
        <mx:Number>14850</mx:Number>
        <mx:Number>53402</mx:Number>
    </MyComp:newStringProperty>
</MyComp:MyTypedArrayComponent>
```

In this example, you try to use Number objects to initialize the Array, so the compiler issues an error.

You can also specify Array properties as tag attributes, rather than using child tags, as the following example shows:

```
<MyComp:MyTypedArrayComponent newNumberProperty="[abc,def]"/>
```

This MXML code generates an error because Flex cannot convert the Strings "`abc`" and "`def`" to a Number.

You insert the `[ArrayElementType]` metadata tag before the variable definition. The tag has the following syntax:

```
[ArrayElementType("elementType")]
```

The following table describes the property of the `[ArrayElementType]` metadata tag:

Property	**Type**	**Description**
`elementType`	String	Specifies the data type of the Array elements, and can be one of the ActionScript data types, such as String, Number, class, or interface. You must specify the type as a fully qualified class name, including the package.

Bindable metadata tag

When a property is the source of a data binding expression, Flex automatically copies the value of the source property to any destination property when the source property changes. To signal to Flex to perform the copy, you must use the `[Bindable]` metadata tag to register the property with Flex, and the source property must dispatch an event.

The `[Bindable]` metadata tag has the following syntax:

```
[Bindable]
[Bindable(event="eventname")]
```

If you omit the event name, Flex automatically creates an event named `propertyChange`.

For more information on data binding and on this metadata tag, see “Binding Data” on page 1225 in *Adobe Flex 3 Developer Guide*.

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.

The following example uses the `[Bindable]` metadata tag for a variable and a getter property. The example also shows 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 specific 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, as the following example shows:

```
dataProvider[i] = newItem
```

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 this problem 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.

DefaultProperty metadata tag

The `[DefaultProperty]` metadata tag defines the name of the default property of the component when you use the component in an MXML file.

The `[DefaultProperty]` metadata tag has the following syntax:

```
[DefaultProperty("propertyName")]
```

The *`propertyName`* property specifies the name of the default property.

You can use the `[DefaultProperty]` metadata tag in your ActionScript component to define a single default property. For more information and an example, see “Creating a default property” on page 113.

Deprecated metadata tag

A class or class elements marked as deprecated is one which is considered obsolete, and whose use is discouraged in the current release. While the class or class element still works, its use can generate compiler warnings.

The mxmlc command-line compiler supports the `show-deprecation-warnings` compiler option, which, when `true`, configures the compiler to issue deprecation warnings when your application uses deprecated elements. The default value is `true`.

Insert the `[Deprecated]` metadata tag before a property, method, or class definition to mark that element as deprecated. The `[Deprecated]` metadata tag has the following options for its syntax when used with a class, property or method:

```
[Deprecated("string_describing_deprecation")]
[Deprecated(message="string_describing_deprecation")]
[Deprecated(replacement="string_specifying_replacement")]
[Deprecated(replacement="string_specifying_replacement",
since="version_of_replacement")]
```

The following uses the `[Deprecated]` metadata tag to mark the `dataProvider` property as obsolete:

```
[Deprecated(replacement="MenuBarItem.data")]
public function set dataProvider(value:Object):void
{
    ...
}
```

The `[Event]`, `[Effect]` and `[Style]` metadata tags also support deprecation. These tags support the following options for syntax:

```
[Event(... , deprecatedMessage="string_describing_deprecation")]
[Event(... , deprecatedReplacement="change2")]
[Event(... , deprecatedReplacement="string_specifying_replacement",
deprecatedSince="version_of_replacement")]
```

These metadata tags support the `deprecatedReplacement` and `deprecatedSince` attributes to mark the event, effect, or style as deprecated.

Effect metadata tag

The `[Effect]` metadata tag defines the name of the MXML property that you use to assign an effect to a component and the event that triggers the effect. If you define a custom effect, you can use the `[Effect]` metadata tag to specify that property to the Flex compiler.

For more information on defining custom effects, see "Effects" on page 199.

An effect is paired with a trigger that invokes the effect. A *trigger* is 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.

You insert the `[Effect]` metadata tag before the class definition in an ActionScript file or in the `<mx:Metadata>` block in an MXML file. The `[Effect]` metadata tag has the following syntax:

```
[Effect(name="eventNameEffect", event="eventName")]
```

The following table describes the properties of the `[Effect]` metadata tag:

Property	**Type**	**Description**
`eventNameEffect`	String	Specifies the name of the effect.
`eventName`	String	Specifies the name of the event that triggers the effect.

The `[Effect]` metadata tag is often paired with an `[Event]` metadata tag, where the `[Event]` metadata tag defines the event corresponding to the effect's trigger. By convention, the name of the effect is the event name with the suffix `Effect`, as the following example of an ActionScript file shows:

```
// Define event corresponding to the effect trigger.
[Event(name="darken", type="flash.events.Event")]
// Define the effect.
[Effect(name="darkenEffect", event="darken")]
class ModalText extends TextArea {
    ...
}
```

In an MXML file, you can define the event and effect in an `<mx:Metadata>` block, as the following example shows:

```
<mx:Metadata>
    [Event(name="darken", type="flash.events.Event")]
    [Effect(name="darkenEffect", event="darken")]
</mx:Metadata>
```

Event metadata tag

Use the `[Event]` metadata tag to define the MXML property for an event and the data type of the event object that a component emits. You insert the [Event] metadata tag before the class definition in an ActionScript file, or in the `<mx:Metadata>` block in an MXML file.

For more information on defining custom events, see "Custom Events" on page 23.

The `[Event]` metadata tag has the following syntax:

```
[Event(name="eventName", type="package.eventType")]
```

The following table describes the properties of the `[Event]` metadata tag:

Property	**Type**	**Description**
`eventName`	String	Specifies the name of the event, including its package name.
`eventType`	String	Specifies the class that defines the data type of the event object. The class name is either the base event class, Event, or a subclass of the Event class. You must include the package in the class name.

The following example identifies the `myClickEvent` event as an event that the component can dispatch:

```
[Event(name="myClickEvent", type="flash.events.Event")]
```

If you do not identify an event in the class file with the `[Event]` metadata tag, the MXML compiler generates an error if you try to use the event name in MXML. Any component can register an event listener for the event in ActionScript by using the `addEventListener()` method, even if you omit the `[Event]` metadata tag.

The following example identifies the `myClickEvent` event as an event that an ActionScript component can dispatch:

```
[Event(name="myEnableEvent", type="flash.events.Event")]
public class MyComponent extends UIComponent
{
    ...
}
```

The following example shows the `[Event]` metadata tag in the `<mx:Metadata>` tag in an MXML file:

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

    <mx:Metadata>
        [Event(name="myEnableEvent", type="flash.events.Event")]
    </mx:Metadata>

    ....

</mx:TextArea>
```

IconFile metadata tag

Use the `[IconFile]` metadata tag to identify the filename for the icon that represents the component in the Insert bar of Flex Builder.

The `[IconFile]` metadata tag has the following syntax:

```
[IconFile("fileName")]
```

The `fileName` property specifies a PNG, GIF, or JPEG file that contains the icon, as the following example shows:

```
[IconFile("MyButton.png")]
public class MyButton extends Button
{
    ...
}
```

Inspectable metadata tag

The `[Inspectable]` metadata tag defines information about an attribute of your component that you expose in code hints and in the Property inspector area of Flex Builder. The `[Inspectable]` metadata tag is not required for either code hints or the Property inspector. The following rules determine how Flex Builder displays this information:

- All public properties in components appear in code hints and in the Flex Builder Property inspector. If you have extra information about the property that you want to add, such as enumeration values or that a String property represents a file path, add the `[Inspectable]` metadata tag with that information.
- Code hints for components and the information in the Property inspector come from the same data. Therefore, if the attribute appears in one, it should appear in the other.
- Code hints for ActionScript components do not require metadata to work correctly so that you always see the appropriate code hints, depending the current scope. Flex Builder uses the `public`, `protected`, `private`, and `static` keywords, plus the current scope, to determine which ActionScript code hints to show.

The `[Inspectable]` metadata tag must immediately precede the property's variable declaration or the setter and getter methods to be bound to that property. The `[Inspectable]` metadata tag has the following syntaxes:

```
[Inspectable(attribute=value[,attribute=value,...])]
property_declaration name:type;

[Inspectable(attribute=value[,attribute=value,...])]
setter_getter_declarations;
```

The following table describes the properties of the `[Inspectable]` metadata tag:

Property	Type	Description
`category`	String	Groups the property into a specific subcategory in the Property inspector of the Flex Builder user interface. The default category is `"Other"`. Specify a value of `"Common"`, `"Effects"`, `"Events"`, `"Layout Constraints"`, `"Size"`, `"Styles"`, or `"Other"`.
`defaultValue`	String or Number	Sets the initial value in the editor that appears in the Property inspector when you modify the attribute. The default value is determined from the property definition.
`enumeration`	String	Specifies a comma-delimited list of legal values for the property. Only these values are allowed; for example, `item1,item2,item3`. Notice the lack of a space character between items so that Flex Builder does not interpret a space as a part of a valid value. This information appears as code hints and in the Property inspector. If you define a Boolean variable, Flex Builder automatically shows `true` and `false` without you having to specifying them using `enumeration`.
`environment`	String	Specifies which inspectable properties should not be allowed (`environment=none`), which are used only for Flex Builder (`environment=Flash`), and which are used only by Flex and not Flex Builder (`environment=MXML`).
`format`	String	Determines the type of editor that appears in the Property inspector when you modify the attribute. You can use this property when the data type of the attribute is not specific to its function. For example, for a property of type Number, you can specify `format="Color"` to cause Flex Builder to open a color editor when you modify the attribute. Common values for the `format` property include `"Length"`, `"Color"`, `"Time"`, `"EmbeddedFile"`, and `"File"`.
`listOffset`	Number	Specifies the default index into a List value.
`name`	String	Specifies the display name for the property; for example, `Font Width`. If not specified, use the property's name, such as `_fontWidth`.

Property	Type	Description
`type`	String	Specifies the type specifier. If omitted, use the property's type. The following values are valid: • `Array` • `Boolean` • `Color` • `Font Name` • `List` • `Number` • `Object` • `String` If the property is an Array, you must list the valid values for the Array.
`variable`	String	Specifies the variable to which this parameter is bound.
`verbose`	Number	Indicates that this inspectable property should be displayed in the Flex Builder user interface only when the user indicates that `verbose` properties should be included. If this property is not specified, Flex Builder assumes that the property should be displayed.

The following example defines the `myProp` parameter as inspectable:

```
[Inspectable(defaultValue=true, verbose=1, category="Other")]
public var myProp:Boolean;
```

InstanceType metadata tag

The `[InstanceType]` metadata tag specifies the allowed data type of a property of type IDeferredInstance, as the following example shows:

```
// Define a deferred property for the top component.
[InstanceType("mx.controls.Label")]
public var topRow:IDeferredInstance;
```

The Flex compiler validates that users assign values only of the specified type to the property. In this example, if the component user sets the `topRow` property to a value of a type other than mx.controls.Label, the compiler issues an error message.

You use the `[InstanceType]` metadata tag when creating template components. For more information, see "Template Components" on page 173.

The `[InstanceType]` metadata tag has the following syntax:

```
[InstanceType("package.className")]
```

You must specify a fully qualified package and class name.

NonCommittingChangeEvent metadata tag

The `[NonCommittingChangeEvent]` metadata tag identifies an event as an interim trigger, which means that the event should not invoke Flex data validators on the property. You use this tag for properties that might change often, but which you do not want to validate on every change.

An example of this is if you tied a validator to the `text` property of a TextInput control. The `text` property changes on every keystroke, but you do not want to validate the property until the user presses the Enter key or changes focus away from the field. The `NonCommittingChangeEvent` tag lets you dispatch a change event, but that does not trigger validation.

You insert the `[NonCommittingChangeEvent]` metadata tag before an ActionScript property definition or before a setter or getter method. The `[NonCommittingChangeEvent]` metadata tag has the following syntax:

```
[NonCommittingChangeEvent("event_name")]
```

In the following example, the component dispatches the `change` event every time the user enters a keystroke, but the `change` event does not trigger data binding or data validators. When the user completes data entry by pressing the Enter key, the component broadcasts the `valueCommit` event to trigger any data bindings and data validators:

```
[Event(name="change", type="flash.events.Event")]
class MyText extends UIComponent {
    ...

    [Bindable(event="valueCommit")]
    [NonCommittingChangeEvent("change")]
    function get text():String {
        return getText();
    }
    function set text(t):void {
        setText(t);
        // Dispatch events.
    }
}
```

RemoteClass metadata tag

Use the `[RemoteClass]` metadata tag to register the class with Flex so that Flex preserves type information when a class instance is serialized by using Action Message Format (AMF). You insert the `[RemoteClass]` metadata tag before an ActionScript class definition. The `[RemoteClass]` metadata tag has the following syntax:

```
[RemoteClass]
```

You can also use this tag 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`. For more information, see "Accessing Server-Side Data with Flex" on page 1159 in *Adobe Flex 3 Developer Guide*.

Style metadata tag

Use the `[Style]` metadata tag to define the MXML tag attribute for a style property for the component. You insert the `[Style]` metadata tag before the class definition in an ActionScript file, or in the `<mx:Metadata>` block in an MXML file.

The `[Style]` metadata tag has the following syntax:

```
[Style(name="style_name"[,property="value",...])]
```

The following table describes the properties for the `[Style]` metadata tag:

Option	**Type**	**Description**
`name`	String	(Required) Specifies the name of the style.
`type`	String	Specifies the data type of the value that you write to the style property. If the type is not an ActionScript type such as Number or Date, use a qualified class name in the form *packageName.className*.
`arrayType`	String	If `type` is `Array`, `arrayType` specifies the data type of the Array elements. If the data type is not an ActionScript type such as Number or Date, use a qualified class name in the form *packageName.className*.
`format`	String	Specifies the units of the property. For example, if you specify `type` as `"Number"`, you might specify `format="Length"` if the style defines a length measured in pixels. Or, if you specify `type="uint"`, you might set `format="Color"` if the style defines an RGB color.

Option	Type	Description
`enumeration`	String	Specifies an enumerated list of possible values for the style property.
`inherit`	String	Specifies whether the property is inheriting. Valid values are `yes` and `no`. This property refers to CSS inheritance, not object-oriented inheritance. All subclasses automatically use object-oriented inheritance to inherit the style property definitions of their superclasses. Some style properties are inherited using CSS inheritance. 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" in *Adobe Flex 3 Developer Guide*.
`states`	String	For skin properties, specifies that you can use the style to specify a stateful skin for multiple states of the component. For example, the definition of the `Slider.thumbSkin` style uses the following `[Style]` metadata tag: `[Style(name="thumbSkin", type="Class", inherit="no", states="disabled, down, over, up")]` This line specifies that you can use the `Slider.thumbSkin` style to specify a stateful skin for the disabled, down, over, and up states of the Slider control. For more information, see "Creating Skins" on page 689 in *Adobe Flex 3 Developer Guide*.

The following example shows the definition of the `textSelectedColor` style property:

```
[Style(name="textSelectedColor",type="Number",format="Color",inherit="yes")]
```

The next example shows the definition of the `verticalAlign` style property:

```
[Style(name="verticalAlign", type="String", enumeration="bottom,middle,top",
inherit="no")]
```

For more information on the `[Style]` metadata tag, see "Custom Style Properties" on page 163.

Transient metadata tag

Use the `[Transient]` metadata tag to identifies a property that should be omitted from data that is sent to the server when an ActionScript object is mapped to a Java object using the `[RemoteClass]` metadata tag.

The `[Transient]` metadata tag has the following syntax:

```
[Transient]
public var count:Number = 5;
```

Chapter 5: Component Compilation

When you compile an application, you create a SWF file that a user can download and play. You can also compile any custom components that you create as part of the application.

When you create a component, you save it to a location that the Adobe® Flex® compiler can access. You can save your components as MXML and ActionScript files, as SWC files, or as Runtime Shared Libraries (RSLs). You have several options when you compile the components.

Topics

About compiling

You compile a custom component so that you can use it as part of your application. You can compile the component when you compile the entire application, or you can compile it separately so that you can link it into the application at a later time.

Flex component file types

When you create a Flex component, you can distribute it in one of several different file formats, as the following table shows:

File format	Extension	Description
MXML	.mxml	A component implemented as an MXML file.
ActionScript	.as	A component implemented as an ActionScript class.
SWC	.swc	A component implemented as an MXML or ActionScript file, and then packaged as a SWC file. A SWC file contains components that you package and reuse among multiple applications. The SWC file is then compiled into your application when you create the application's SWF file.
RSL	.swc	A component implemented as an MXML or ActionScript file, and then deployed as an RSL. An RSL is a stand-alone file that is downloaded separately from your application's SWF file, and cached on the client computer for use with multiple application SWF files.

You must take into consideration the file format and file location when you compile an application that uses the component.

About compiling with Flex SDK

Adobe Flex includes two compilers: mxmlc and compc. You use the mxmlc compiler to compile MXML, ActionScript, SWC, and RSL files into a single SWF file. After your application is compiled and deployed on your web or application server, a user can make an HTTP request to download and play the SWF file on their computer.

You use the compc compiler to compile components, classes, and other files into SWC files or into RSLs.

You can use the compc and mxmlc compilers from Adobe Flex Builder or from a command line. For more information on using the compilers, see "Using the Flex Compilers" on page 125 in *Building and Deploying Adobe Flex 3 Applications*, and "Building Projects" on page 121 in *Using Flex Builder*.

The most basic example of using the mxmlc compiler is one in which the MXML file has no external dependencies (such as components in a SWC file or ActionScript classes). In this case, you open mxmlc and point it to your MXML file:

```
$ mxmlc c:/myfiles/app.mxml
```

The default option is the target file to compile into a SWF file, and it must have a value. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file, as in the following example:

```
$ mxmlc -option arg1 arg2 arg3 -- target_file.mxml
```

About case sensitivity during a compilation

The Flex compilers use a case-sensitive file lookup on all file systems. On case-insensitive file systems, such as the Macintosh and Windows file systems, the Flex compiler generates a case-mismatch error when you use a component with the incorrect case. On case-sensitive file systems, such as the UNIX file system, the Flex compiler generates a component-not-found error when you use a component with the incorrect case.

About the ActionScript source path

Typically, you put component files in directories that are in the ActionScript source path. These include your application's root directory, its subdirectories, and any directory that you specify to the compiler. To specify a directory, you use the `source-path` option to the mxmlc compiler, or the Project Properties dialog box in Flex Builder.

The following rules can help you organize your custom components:

- An application can access MXML and ActionScript components in the same directory and in its subdirectories.
- An ActionScript component in a subdirectory of the main application directory must define a fully qualified package name that is relative to the location of the application's root directory. For example, if you define a custom component in the dir1/dir2/myControls/PieChart.as file, its fully qualified package name must be dir1.dir2.myControls, assuming dir1 is an immediate subdirectory of the main application directory.
- An MXML component does not include a package name definition. However, you must declare a namespace definition in the file that references the MXML component that corresponds to the directory location of the MXML component, either in a subdirectory of the application's root directory or in a subdirectory of the source path. For more information, see "Simple MXML Components" on page 63.
- An application can access MXML and ActionScript components in the directories included in the ActionScript source path. The component search order in the source path is based on the order of the directories listed in the source path.
- An ActionScript component in a subdirectory of a directory included in the source path must define a fully qualified package name that is relative to the location of the source path directory. For example, if you define a custom component in the file dir1/dir2/myControls/PieChart.as, and dir1 is included in the ActionScript source path, its fully qualified package name must be dir1.dir2.myControls.
- The `<mx:Script>` tag in the main MXML file, and in dependent MXML component files, can reference components located in the ActionScript source path.

Compiling components with Flex SDK

How you compile an application with Adobe Flex SDK is based on how you distribute your custom components that are distributed as MXML, ActionScript, SWC, and RSL files.

Distributing components as MXML and ActionScript files

When you compile an application with Flex SDK, you define where the MXML and ActionScript files for your custom components exist in the directory structure of your application, or in the directory structure of components shared by multiple applications.

For example, you can create a component for use by a single application. In that case, you store it in the directory structure of the application, usually in a subdirectory under the directory that contains the main file of the application. The component is then compiled with the entire application into the resultant SWF file.

You can also create a component that is shared among multiple applications as an MXML or ActionScript file. In that case, store the component in a location that is included in the ActionScript source path of the application. When Flex compiles the application, it also compiles the components included in the application's ActionScript source path.

You specify the directory location of the shared components by using one of the following methods:

Flex Builder Open the Project Properties dialog box, and then select Flex Build Path to set the ActionScript source path.

mxmlc compiler Use the `source-path` option to the mxmlc compiler to specify the directory location of your shared MXML and ActionScript files.

Distributing components as SWC files

A SWC file is an archive file of Flex components. SWC files make it easy to exchange components among Flex developers. You need to exchange only a single file, rather than the MXML or ActionScript files, images, and other resource files. In addition, the SWF file inside a SWC file is compiled, which means that the code is hidden from casual view. Finally, compiling a component as a SWC file can make namespace allocation an easier process.

SWC files can contain one or more components and are packaged and expanded with the PKZIP archive format. You can open and examine a SWC file by using WinZip, JAR, or another archiving tool. However, do not manually change the contents of a SWC file, and do not try to run the SWF file that is in a SWC file outside of the SWC file.

When you compile your application, you specify the directory location of the SWC files by using one of the following methods:

Flex Builder Open the Project Properties dialog box, and then select Flex Build Path to set the library directories that contain the SWC files.

mxmlc compiler Set the `library-classpath` option to the mxmlc compiler to specify the directory location of your SWC files.

One of the advantages of distributing components as SWC files is that you can define a global style sheet, named defaults.css, in the SWC file. The defaults.css file defines the default style settings for all of the components defined in the SWC file. For more information, see "Applying styles from a defaults.css file" on page 74.

For more information about SWC files, see "Using the Flex Compilers" on page 125 in *Building and Deploying Adobe Flex 3 Applications*, and "Building Projects" on page 121 in *Using Flex Builder*.

Distributing components as RSLs

One way to reduce the size of your application's SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but must be transferred to the client only once. These shared files are known as Runtime Shared Libraries (RSLs).

If you have multiple applications but those applications share a core set of components or classes, your users download those assets only once as an RSL. The applications that share the assets in the RSL use the same cached RSL as the source for the libraries as long as they are in the same domain. By using an RSL, you can reduce the resulting file size for your applications. The benefits increase as the number of applications that use the RSL increases. If you only have one application, putting components into RSLs does not reduce the aggregate download size, and may increase it.

When you compile your application, you specify the directory location of an RSL file by using one of the following methods:

Flex Builder Open the Project Properties dialog box, and then select Flex Build Path to set the library directories that contain the SWC files.

mxmlc compiler Set the `external-library-path` option to the mxmlc compiler to specify the location of the RSL file at compile time. Set the `runtime-shared-libraries` option to the mxmlc compiler to specify the relative location of the RSL file when the application is deployed.

For more information, including information on how to create an RSL file, see “Using Runtime Shared Libraries” on page 195 in *Building and Deploying Adobe Flex 3 Applications*.

Example: Compiling a custom formatter component

The example in this section uses a custom formatter component that is defined as an ActionScript file. The name of the formatter is MySimpleFormatter, and it is defined in the file MySimpleFormatter.as. For more information on creating customer formatter components, see “Custom Formatters” on page 183.

The process for compiling an MXML file is the same as for an ActionScript file. For an example of deploying an MXML file, see “Simple MXML Components” on page 63.

Distributing a component as an ActionScript file

When you distribute a component defined as an ActionScript file, you can store it within the same directory structure as your application files or in a directory specified in the ActionScript source path.

The MXML tag name for a custom component consists of two parts: the namespace prefix and the tag name. The namespace prefix tells Flex where to look for the file that implements the custom component. The tag name corresponds to the filename of the component, in this case MySimpleFormatter.as. Therefore, the file MySimpleFormatter.as defines a component with the tag name of `<`*`namespace`*`:MySimpleFormatter>`.

The main application MXML file defines the namespace prefix used to reference the component in the `<mx:Application>` tag. When you deploy your formatter as an ActionScript file, you refer to it in one of the following ways:

1 If you store the formatter component in the same directory as the application file, or in a directory that the ActionScript source path (not a subdirectory) specifies, you define the formatter by using an empty `package` statement, as the following example shows:

```
package
{
    //Import base Formatter class.
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

You can refer to it as the following example shows. In the following code, the local namespace (*) is mapped to the prefix MyComp.

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

    <MyComp:MySimpleFormatter/>

</mx:Application>
```

If the same file exists in the ActionScript source path directory and the application directory, Flex uses the file in the application directory.

2 If you store the formatter component in a subdirectory of the directory that contains the application file, you specify that directory as part of the `package` statement, as the following example shows:

```
package myComponents.formatters
{
    //Import base Formatter class
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

In this example, the MySimpleFormatter.as file is located in the myComponents/formatter subdirectory of the main application directory. You map the myComponents.formatters namespace to the `MyComp` prefix, as the following example shows:

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

    <MyComp:MySimpleFormatter/>

</mx:Application>
```

If multiple files with the same name exist under an ActionScript source path subdirectory and the application subdirectory, Flex uses the file under the application subdirectory.

3 If you store the formatter component in a subdirectory of the ActionScript source path directory, you specify that subdirectory as part of the `package` statement, as the following example shows:

```
package flexSharedRoot.custom.components
{
    //Import base Formatter class.
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

You then use a namespace that specifies the subdirectory. The following code declares a component that is in the flexSharedRoot/custom/components directory:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:MyComp="flexSharedRoot.custom.components.*"/>

    <MyComp:MySimpleFormatter/>

</mx:Application>
```

If the same file exists in the ActionScript source path directory and the application directory, Flex uses the file in the application file directory.

Distributing a component as a SWC file

To create a SWC file, use the compc compiler in the *flex_install_dir*/bin directory. The compc compiler generates a SWC file from MXML component source files and/or ActionScript component source files.

In this example, you create a SWC file for a custom formatter component that you defined by using the following package and class definition:

```
package myComponents.formatters
{
    //Import base Formatter class.
    import mx.formatters.Formatter

    public class MySimpleFormatter extends Formatter {
        ...
    }
}
```

In this example, the MySimpleFormatter.as file is in the directory c:\flex\myComponentsForSWCs\myComponents\formatters.

You use the following compc command from the *flex_install_dir*/bin directory to create the SWC file for this component:

```
.\compc -source-path c:\flex\myComponentsForSWCs\
    -include-classes myComponents.formatters.MySimpleFormatter
    -o c:\flex\mainApp\MyFormatterSWC.swc
```

In this example, you use the following options of the compc compiler:

Option name	Description
`-source-path`	Specifies the base directory location of the MySimpleFormatter.as file. It does not include the directories that the component's package statement defines.
`-include-classes`	Specifies classes to include in the SWC file. You provide the class name (MyComponents.formatters.MySimpleFormatter) not the filename of the source code. All classes specified with this option must be in the compiler's source path, which is specified in the `source-path` compiler option. You can use packaged and unpackaged classes. To use components in namespaces, use the `include-namespaces` option. If the components are in packages, use dot (.) notation rather than slashes to separate package levels.
`-o`	Specifies the name and directory location of the output SWC file. In this example, the directory is c:\flex\mainApp, the directory that contains your main application.

In your main application file, you specify the component's namespace, as the following example shows:

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

    <MyComp:MyFormatter/>

</mx:Application>
```

When you distribute SWC files, ensure that the corresponding ActionScript file is not in the directory structure of the application or in the ActionScript source path. Otherwise, Flex might use the ActionScript file, rather than the SWC file.

When you use mxmlc to compile the main application, ensure that the c:\flex\mainApp directory is included in the library path; otherwise, mxmlc cannot locate the SWC file.

For more information about SWC files, see "Using the Flex Compilers" on page 125 in *Building and Deploying Adobe Flex 3 Applications*, and "Building Projects" on page 121 in *Using Flex Builder*.

Distributing a component as an RSL file

You create an RSL by using the compc tool, and then pass the library's location to the compiler when you compile your application. For more information, including an example, see "Using Runtime Shared Libraries" on page 195 in *Building and Deploying Adobe Flex 3 Applications*.

Part 2: MXML Custom Components

Topics

Chapter 6: Simple MXML Components

Adobe® Flex® applications typically consist of multiple MXML and ActionScript files, and each MXML file is a separate MXML component. MXML components let you encapsulate functionality in a reusable component, extend an existing Flex component by adding new functionality to it, and reference the MXML component by using an MXML tag.

For information on advanced techniques for creating MXML components, see "Advanced MXML Components" on page 79.

Topics

About MXML components

In typical Flex applications, you do not code the entire application within a single source code file. Such an implementation makes it difficult for multiple developers to work on the project simultaneously, makes it difficult to debug, and discourages code reuse.

Instead, you develop Flex applications using multiple MXML and ActionScript files. This architecture promotes a modular design, code reuse, and lets multiple developers contribute to the implementation.

MXML components are MXML files that you reference by using MXML tags from within other MXML files. One of the main uses of MXML components is to extend the functionality of an existing Flex component.

For example, Flex supplies a ComboBox control that you can use as part of a form that collects address information from a customer. You can use a ComboBox to let the user select the State portion of the address from a list of the 50 states in the U.S. In an application that has multiple locations where a user can enter an address, it would be tedious to create and initialize multiple ComboBox controls with the information about all 50 states.

Instead, you create an MXML component that contains a ComboBox control with all 50 states defined within it. Then, wherever you must add a state selector to your application, you use your custom MXML component.

Creating MXML components

An application that uses MXML components includes a main MXML application file, which contains the `<mx:Application>` root tag, and references one or more components that are defined in separate MXML and ActionScript files. Each MXML component extends an existing Flex component, or another MXML component.

You create an MXML component in an MXML file where the component's filename becomes its MXML tag name. For example, a file named StateComboBox.mxml defines a component with the tag name of `<StateComboBox>`.

The root tag of an MXML component is a component tag, either a Flex component or another MXML component. The root tag specifies the http://www.adobe.com/2006/mxml namespace. For example, the following MXML component extends the standard Flex ComboBox control.

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

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

    <mx:dataProvider>
        <mx:String>AK</mx:String>
        <mx:String>AL</mx:String>
        <!-- Add all other states. -->
        </mx:dataProvider>
</mx:ComboBox>
```

As part of its implementation, a custom MXML component can reference another custom MXML component.

The main application, or any other MXML component file, references the StateComboBox component, as the following example shows:

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

    <MyComp:StateComboBox/>

</mx:Application>
```

In this example, the main application file includes a new namespace definition of `xmlns:MyComp="*"` as part of the `<mx:Application>` tag. This namespace definition specifies the location of the MXML component. In this case, it specifies that the component is in the same directory as the main application file, or if you are using Adobe® LiveCycle™ Data Services ES, in the WEB-INF/flex/user-classes directory.

As a best practice, store your components in a subdirectory. For example, you can write this file to the myComponents directory, a subdirectory of your main application directory. For more information on the namespace, see "Developing Applications in MXML" on page 3 in *Adobe Flex 3 Developer Guide*.

The StateComboBox.mxmlfile specifies the ComboBox control as its root tag, so you can reference all of the properties of the ComboBox control within the MXML tag of your custom component, or in the ActionScript specified within an `<mx:Script>` tag. For example, the following code specifies the `rowCount` property and a listener for the `close` event for your custom control:

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

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private function handleCloseEvent(eventObj:Event):void {
                myTA.text="foo";

            }
        ]]>
    </mx:Script>

    <local:StateComboBox rowCount="5" close="handleCloseEvent(event);"/>

    <mx:TextArea id="myTA" />

</mx:Application>
```

Limitation on the default property of the root tag

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>
```

However, the default property mechanism does not work for root tags of MXML components. In this situation, you must use child tags to define the default property, as the following example shows:

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

MXML components and ActionScript classes

When you create a custom MXML component, you define a new ActionScript class where the class name corresponds to the filename of the MXML component. Your new class is a subclass of the component's root tag, and therefore inherits all of the properties and methods of the root tag. However, because you are defining the component in MXML, many of the intricacies of creating an ActionScript class are hidden from you.

For example, in "Creating MXML components" on page 64, you defined the component StateComboBox.mxml by using the `<mx:ComboBox>` tag as its root tag. Therefore, StateComboBox.mxml defines a subclass of the ComboBox class.

Creating composite MXML components

A composite MXML component is a component that contains multiple component definitions within it. To create a composite component, you specify a container as its root tag, and then add Flex components as children of the container.

For example, the following component contains an address form created by specifying a Form container as the root tag of the component, and then defining several children of the Form container. One of the `<mx:FormItem>` tags contains a reference to the `<MyComp:StateComboBox>` tag that you created in "Creating MXML components" on page 64:

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

<mx:Form xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:MyComp="*">

    <mx:FormItem label="NameField">
        <mx:TextInput/>
    </mx:FormItem>

    <mx:FormItem label="Street">
        <mx:TextInput/>
    </mx:FormItem>
    <mx:FormItem label="City" >
        <mx:TextInput/>
    </mx:FormItem>

    <mx:FormItem label="State" >
        <MyComp:StateComboBox/>
    </mx:FormItem>

</mx:Form>
```

The following application file references the AddressForm component in the `<AddressForm>` tag:

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

    <MyComp:AddressForm/>

</mx:Application>
```

If you include child tags of the root container tag in an MXML component file, you cannot add child tags when you use the component as a custom tag in another MXML file. If you define an empty container in an MXML file, you can add child tags when you use the component as a custom tag.

Note: *The restriction on child tags refers to the child tags that correspond to visual components. Visual components are subclasses of the UIComponent component. You can always insert tags for nonvisual components, such as ActionScript blocks, styles, effects, formatters, validators, and other types of nonvisual components, regardless of how you define your custom component.*

The following example defines an empty Form container in an MXML component:

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

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

This component defines no children of the Form container; therefore, you can add children when you use it in another MXML file, as the following example shows:

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

    <MyComp:EmptyForm>
        <mx:FormItem label="Name">
            <mx:TextInput/>
        </mx:FormItem>
    </MyComp:EmptyForm>

</mx:Application>
```

The AddressForm.mxml file specifies the Form container as its root tag. Because you define a container as the root tag of the MXML component, you are creating a subclass of that container, and you can reference all of the properties and methods of the root tag when using your MXML component. Therefore, in the main application, you can reference all of the properties of the Form container in the MXML tag that corresponds to your custom component, or in any ActionScript code in the main application. However, you cannot reference properties of the children of the Form container.

For example, the following example sets the `horizontalPageScrollSize` property and a listener for the `scroll` event for your custom control, but you cannot specify properties for the child CheckBox or TextInput controls of the Form container:

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

    <mx:Script>
        <![CDATA[
            import mx.events.ScrollEvent;

            private function handleScrollEvent(event:ScrollEvent):void {
                // Handle scroll event.
            }
        ]]>
    </mx:Script>

    <MyComp:AddressForm horizontalPageScrollSize="25"
scroll="handleScrollEvent(event);"/>

</mx:Application>
```

To configure the children of a custom MXML component, you define new properties in the MXML component, and then use those new properties to pass configuration information to the component children. For more information, see “Advanced MXML Components” on page 79.

Scoping in custom components

Scoping is mostly a description of what the `this` keyword refers to at any given point in your application. In an `<mx:Script>` tag in an MXML file, the `this` keyword always refers to the current scope. In the main application file, the file that contains the `<mx:Application>` tag, the current scope is the Application object; therefore, the `this` keyword refers to the Application object.

In an MXML component, Flex executes in the context of the custom component. The current scope is defined by the root tag of the file. So, the `this` keyword refers not to the Application object, but to the object defined by the root tag of the MXML file.

For more information on scoping, see "Using ActionScript" on page 37 in *Adobe Flex 3 Developer Guide*.

The root tag of an MXML component cannot contain an `id` property. Therefore, if you refer to the object defined by the root tag in the body of the component, you must use the `this` keyword, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/myComponents/StateComboBoxThis.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    close="handleCloseEvent(event);">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            // Define a property to hold the current index.
            public var stateIndex:Number;

            private function handleCloseEvent(eventObj:Event):void {
                stateIndex = this.selectedIndex;
            }
        ]]>
    </mx:Script>

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

This example defines an event listener for the ComboBox control that updates the `stateIndex` property when the ComboBox control closes.

Applying styles to your custom component

Along with skins, styles define the look and feel of your Flex applications. You can use styles to change the appearance of a single component, or apply them across all components.

When working with custom components, you have several options for how you use styles. You can define your custom components so that they contain no style information at all. That design allows the application developer who is using your component to apply styles to match the rest of their application. For example, if you define a custom component to display text, the application developer can style it to ensure that the font, font size, and font style of your component match the rest of the application.

Alternatively, you might develop a component that you want to deploy with a built-in look so that it is not necessary for application developers to apply any additional styles to it. This type of component might be useful for applications that require a header or footer with a fixed look, while the body of the application has more flexibility in its look.

Or you might develop a custom component by using a combination of these approaches. This type of design lets application developers set some styles, but not others.

For general information on Flex styles, see “Using Styles and Themes” on page 589 in *Adobe Flex 3 Developer Guide*. For information on creating custom styles, see “Custom Style Properties” on page 163.

Applying styles from the custom component

You can choose to define styles within your MXML component so that the component has the same appearance whenever it is used, and application developers do not have to worry about applying styles to it.

In the definition of your custom component, you define styles by using one or both of the following mechanisms:

- Tag properties
- Class selectors

The following custom component defines a style by using a tag property of the ComboBox control:

```
<?xml version="1.0"?>
<!-- mxml/myComponents/StateComboBoxWithStyleProps.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    openDuration="1000"
    fontSize="15">

    <mx:dataProvider>
        <mx:Array>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
        </mx:Array>
    </mx:dataProvider>
</mx:ComboBox>
```

Alternatively, you can define these styles by using a class selector style declaration, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/myComponents/StateComboBoxWithStyleClassSel.mxml -->

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

    <mx:Style>
        .myCBStyle {
            openDuration : 1000;
            fontSize : 15;
            }
    </mx:Style>

    <mx:dataProvider>
        <mx:Array>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
        </mx:Array>
    </mx:dataProvider>
</mx:ComboBox>
```

Note: *You cannot define a type selector in an MXML component. If you define a type selector, a compiler error occurs.*

Application developers can apply additional styles to the component. For example, if your component defines styles for the open duration and font size, application developers can still specify font color or other styles. The following example uses StateComboBoxWithStyleProps.mxml in an application and specifies the font color style for the control:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleWithPropsAddColor.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateComboBoxWithStyleProps color="red"/>

</mx:Application>
```

Applying styles from the referencing file

When you reference an MXML component, the referencing file can specify style definitions to the MXML component by using the following mechanisms:

- Tag properties
- Class selectors
- Type selectors

The styles that application developers can apply correspond to the styles supported by the root tag of the MXML component. The following example uses a tag property to set a style for the custom MXML component:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleWithPropsOverrideOpenDur.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateComboBoxWithStyleProps openDuration="1000"/>

</mx:Application>
```

When you specify styles as tag attributes, those styles override any conflicting styles set in the definition of the MXML component.

You can use a class selector to define styles. Often you use a class selector to apply styles to specific instances of a control, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleOverrideUsingClassSel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Style>
        .myStateComboBox {
            openDuration : 1000;
        }
    </mx:Style>

    <MyComp:StateComboBoxWithStyleProps styleName="myStateComboBox"/>
    <mx:ComboBox>
        ...
    </mx:ComboBox>
</mx:Application>
```

In this example, you use the `styleName` property in the tag definition of an MXML component to apply styles to a specific instance of the MXML component. However, those styles are not applied to the ComboBox control defined in the main application file, nor would they be applied to any other instances of StateComboBox.mxml unless you also specify the `styleName` property as part of defining those instances of the MXML component.

When you specify any styles by using a class selector, those styles override all styles that you set by using a class selector in the MXML file. Those styles do not override styles that you set by using tag properties in the MXML file.

You can also use a type selector to define styles. A type selector applies styles to all instances of a component, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleOverrideUsingTypeSel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Style>
        StateComboBoxWithStyleProps {
            openDuration : 1000;
        }
    </mx:Style>

    <MyComp:StateComboBoxWithStyleProps/>
</mx:Application>
```

In this example, the type selector specifies the `openDuration` style for all instances of the StateComboBox control in the application. When you specify any styles by using a type selector, those styles override all styles that you set by using a class selector in the MXML file. Those styles do not override styles that you set by using tag properties in the MXML file.

Applying a type selector to the root tag of a custom component

All custom components contain a root tag that specifies the superclass of the component. In the case of StateComboBox.mxml, the root tag is `<mx:ComboBox>`. If you define a type selector for the ComboBox control, or for a superclass of the ComboBox control, in your main application file, that style definition is also applied to any custom component that uses a ComboBox control as its root tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleOverrideUsingCBTypeSel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Style>
        ComboBox {
            openDuration: 1000;
            fontSize: 15;
            color: red;
        }
    </mx:Style>

    <MyComp:StateComboBoxWithStyleProps/>
    <mx:ComboBox/>

</mx:Application>
```

In this example, all ComboBox controls and all StateComboBox.mxml controls have an `openDuration` of 1000 ms, `fontSize` of 15 points, and red text.

If you define a type selector for a superclass of the custom control and for the custom control itself, Flex ignores any conflicting settings from the type selector for the superclass, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxml/MainStyleOverrideUsingCBTypeSelConflict.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Style>
        ComboBox {
            color: red;
            openDuration: 1000;
            fontSize: 15;
        }
        StateComboBoxWithStyleProps {
            color: green;
        }
    </mx:Style>

    <MyComp:StateComboBoxWithStyleProps/>
    <mx:ComboBox/>
</mx:Application>
```

In this example, the StateComboBox control uses green text, and the values for the `fontSize` and `openDuration` styles specified in the type selector for the ComboBox control.

Applying styles from a defaults.css file

Flex includes a global style sheet, named defaults.css, inside the framework.swc file in the /frameworks/libs directory. This global style sheet contains style definitions for the global class selector and type selectors for most Flex components. Flex implicitly loads the defaults.css file and applies it to all Flex applications during compilation.

One of the most common ways for you to distribute your custom MXML and ActionScript components is to create a SWC file. A SWC file is an archive file of Flex components that make it easy to exchange components among Flex developers. You need to exchange only a single file, rather than the MXML or ActionScript files, images, and other resource files. The SWF file inside a SWC file is compiled, which means that the code is hidden from casual view.

For more information about SWC files, see “Using the Flex Compilers” on page 125 in *Building and Deploying Adobe Flex 3 Applications* and “Building Projects” on page 121 in *Using Flex Builder*.

A SWC file can contain a local style sheet, named defaults.css, that contains the default style settings for the custom components defined within the SWC file. When Flex compiles your application, it automatically applies the local defaults.css to the components in your SWC file. In this way, you can override the global defaults.css style sheet with the local version in your SWC file.

The only requirements on the local version are:

- You can include only a single style sheet in the SWC file.
- The file must be named defaults.css.
- The file must be in the top-most directory of the SWC file.

For example, you define a custom ComboBox control, named StateComboBox.mxml, as the following example shows:

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

    <mx:dataProvider>
        <mx:String>AK</mx:String>
        <mx:String>AL</mx:String>
        <!-- Add all other states. -->
        </mx:dataProvider>
</mx:ComboBox>
```

Notice that this control does not define any style settings.

Then you create a defaults.css file in the same directory to define the default style settings for the custom component, as the following example shows:

```
StateComboBox
{
    arrowButtonWidth: 40;
    cornerRadius: 10;
    fontSize: "14";
    fontWeight: "bold";
    leading: 0;
    paddingLeft: 10;
    paddingRight: 10;
}
```

To create the SWC file, you run the compc command-line compiler from the directory that contains the defaults.css and StateComboBox.mxml files. Use the `include-file` option to specify the style sheet, as the following example shows:

```
flex_install_dir\bin\compc -source-path .
    -include-classes StateComboBox
    -include-file defaults.css defaults.css
    -o MyComponentsSWC.swc
```

This example creates a SWC file named MyComponentsSWC.swc.

***Note:** You can also use the `include-stylesheet` option to include the style sheet if the style sheet references assets that must be compiled, such as programmatic skins or other class files. For more information, see “Using the Flex Compilers” on page 125 in Building and Deploying Adobe Flex 3 Applications.*

Then you write an application, named DefaultCSSApplication.mxml, that uses the StateComboBox control, as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/DefaultCSSApplication.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="*">

    <mx:Label text="Custom MXML component using defaults.css from SWC file."/>
    <MyComp:StateComboBox/>

    <mx:Label text="Default ComboBox control using the default defaults.css file."/>
    <mx:ComboBox>
        <mx:dataProvider>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
        </mx:dataProvider>
    </mx:ComboBox>
</mx:Application>
```

Notice that you include a namespace definition for the StateComboBox control in DefaultCSSApplication.mxml to reference the component. Because the MXML component is in the top-most directory of the SWC file, its package name is blank, and you reference it by using a namespace definition of `xmlns:MyComp="*"`.

To compile an application that uses MyComponentsSWC.swc, copy MyComponentsSWC.swc to the directory that contains the application. Then add the SWC file to the Library Path in Adobe® Flex® Builder™, or use the `-library-path` option to the mxmlc command-line compiler, as the following example shows:

```
flex_install_dir\bin\mxmlc
    -file-specs DefaultCSSApplication.mxml
    --library-path+=.
```

In the previous example, the defaults.css file and the component file were in the same directory. Typically, you place components in a directory structure, where the package name of the component reflects the directory location of the component in the SWC file.

In the next example, you put the StateComboBox.mxml file in the myComponents subdirectory of the SWC file, where the subdirectory corresponds to the package name of the component. However, defaults.css must still be in the top-level directory of the SWC file, regardless of the directory structure of the SWC file. You compile this SWC file by using the following command line:

```
flex_install_dir\bin\compc -source-path .
    -include-classes myComponents.StateComboBox
    -include-file defaults.css defaults.css
    -o MyComponentsSWC.swc
```

To use the StateComboBox.mxml component in your application, you define DefaultCSSApplication.mxml as the following example shows:

```
<?xml version="1.0"?>
<!-- styles/DefaultCSSApplicationSubDir.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Label text="Custom MXML component using defaults.css from SWC file."/>
    <MyComp:StateComboBox/>

    <mx:Label text="Default ComboBox control using the default defaults.css file."/>
    <mx:ComboBox>
        <mx:dataProvider>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
        </mx:dataProvider>
    </mx:ComboBox>
</mx:Application>
```

Notice that the namespace definition for the StateComboBox control in DefaultCSSApplication.mxml includes `myComponents.*` to match the directory structure, and package name, of the component in the SWC file.

You can modify and test your defaults.css file without having to recreate the SWC file by using the `-defaults-css-files` option to the compiler. The CSS files added by the `-defaults-css-files` option have a higher precedence than those in SWC files, so that they can override a corresponding definition in a SWC file.

When you are done modifying the defaults.css file, recreate the SWC file with the updated defaults.css file.

Chapter 7: Advanced MXML Components

One of the common goals of creating MXML components is to create configurable and reusable components. For example, you might want to create MXML components that take properties, dispatch events, define new style properties, have custom skins, or use other customizations.

For information about how to create and deploy simple MXML components, including how to apply styles and skins to your MXML components, see "Simple MXML Components" on page 63.

Topics

About reusable MXML components

One design consideration when you create custom MXML components is reusability. That is, do you want to create a component that is tightly coupled to your application, or one that is reusable in multiple applications?

A tightly coupled component is written for a specific application, and is often dependent on the application's structure, variable names, or other details. If you change the application, you will probably need to modify the component to reflect that change, and it will also be difficult to use the component in another application without rewriting it.

Another possibility is to design a loosely coupled component for reuse. A loosely coupled component has a well-defined interface that specifies how to pass information to the component, and how the component passes back results to the application.

With loosely coupled components, you typically define properties of the component to let users pass information to it. These properties, defined by using variables or setter and getter methods, specify the data types of parameter values. For more information about defining component properties, see "Adding custom properties and methods to a component" on page 80.

The best practice for defining components that return information to the main application is to design the component to dispatch an event that contains the return data. In that way, the main application can define an event listener to handle the event and take the appropriate action. For more information on dispatching events, see "Working with events" on page 93.

Adding custom properties and methods to a component

MXML components provide you with a simple way to create ActionScript classes. When defining classes, you use class properties to store information and class methods to define class functionality. When creating MXML components, you can also add properties and methods to the components to make them configurable. By allowing the user to pass information to the components, you can create a reusable component that you can use in multiple locations throughout your application, or in multiple applications.

Defining properties and methods in MXML components

You can define methods for your MXML components in ActionScript, and properties in ActionScript or MXML. The Adobe Flex compiler converts the MXML component into an ActionScript class, so there is no performance difference between defining a property in MXML and defining it in ActionScript.

Defining properties and methods in ActionScript

With ActionScript, you define properties and methods by using the same syntax that you use in an ActionScript class. For more information on using ActionScript to define properties and methods, see "Custom ActionScript Components" on page 13.

When using ActionScript, you place a property or method definition within an `<mx:Script>` block. The `<mx:Script>` tag must be an immediate child tag of the root tag of the MXML file. A public variable declaration or a set function in an `<mx:Script>` tag becomes a property of the component. A public ActionScript function in an `<mx:Script>` tag becomes a method of the component.

In the following example, the component defines two data providers to populate the ComboBox control, and a function to use as the event listener for the `creationComplete` event. This function sets the data provider of the ComboBox based on the value of the `shortNames` property. By default, the `shortNames` property is set to `true`, to display two-letter names.

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPropAS.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="setNameLength();">

    <mx:Script>
        <![CDATA[

            // Define public variables.
            public var shortNames:Boolean = true;

            // Define private variables.
            private var stateArrayShort:Array = ["AK", "AL"];
            private var stateArrayLong:Array = ["Arkansas", "Alaska"];

            // Define listener method.
            public function setNameLength():void {
                if (shortNames) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
            }
        ]]>
    </mx:Script>
</mx:ComboBox>
```

The following MXML application file uses the `<MyComp:StateComboBoxPropAS>` tag to configure the control to display long state names:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateComboBoxPropAS shortNames="true"/>

</mx:Application>
```

The following example modifies the component to add a method that lets you change the display of the state name at run time. This public method takes a single argument that specifies the value of the `shortNames` property.

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPropMethod.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="setNameLength();">
    <mx:Script>
        <![CDATA[

            // Define public variables.
            public var shortNames:Boolean = true;

            // Define private variables.
            private var stateArrayShort:Array = ["AK", "AL"];
            private var stateArrayLong:Array = ["Arkansas", "Alaska"];

            public function setNameLength():void {
                if (shortNames) {
                    this.dataProvider=stateArrayShort; }
                else {
                    this.dataProvider=stateArrayLong; }
            }

            public function setShortName(val:Boolean):void {
                shortNames=val;
                if (val) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
            }
        ]]>
    </mx:Script>
</mx:ComboBox>
```

You might use this new method with the `click` event of a Button control to change the display from long names to short names, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropWithMethod.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateComboBoxPropMethod id="myCB" shortNames="false"/>
    <mx:Button label="Use Short Names" click="myCB.setShortName(true);"/>

</mx:Application>
```

Defining properties in MXML

In MXML, you can use an MXML tag to define a property of any type, as long as the type refers to an ActionScript class name. For example, you can use the `<mx:String>`, `<mx:Number>`, and `<mx:Boolean>` tags to define properties in your MXML components that take String, Number, or Boolean values, respectively. When using one of these tags, you must specify an `id`, which becomes the property name.

Optionally, you can specify an initial value in the body of the tag, or you can use the `source` property to specify the contents of an external URL or file as the initial property value. If you use the `source` property, the body of the tag must be empty. The initial value can be static data or a binding expression.

The following examples show initial properties set as static data and binding expressions. Values are set in the tag bodies and in the `source` properties.

```
<!-- Boolean property examples: -->
<mx:Boolean id="myBooleanProperty">true</mx:Boolean>
<mx:Boolean id="passwordStatus">{passwordExpired}</mx:Boolean>

<!-- Number property examples: -->
<mx:Number id="myNumberProperty">15</mx:Number>
<mx:Number id="minutes">{numHours * 60}</mx:Number>

<!-- String property examples: -->
<mx:String id="myStringProperty">Welcome, {CustomerName}.</mx:String>
<mx:String id="myStringProperty1" source="./file"/>
```

All properties defined by using the `<mx:String>`, `<mx:Number>`, and `<mx:Boolean>` tags are public. This means that the component user can access these properties.

The following example modifies the example in “Defining properties and methods in ActionScript” on page 80 to define the `shortNames` property by using an MXML tag, rather than an ActionScript variable definition:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPropMXML.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="setNameLength();">

    <!-- Control display of state names. -->
    <mx:Boolean id="shortNames">true</mx:Boolean>

    <mx:Script>
        <![CDATA[

            // Define private variables.
            private var stateArrayShort:Array = ["AK", "AL"];
            private var stateArrayLong:Array = ["Arkansas", "Alaska"];
            // Define listener method.
            public function setNameLength():void {
                if (shortNames) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
            }
        ]]>
    </mx:Script>
</mx:ComboBox>
```

In the preceding example, you implement the StateComboBox.mxml file by using the `<mx:Boolean>` tag to add a new property, `shortNames`, with a default value of `true`. This property controls whether the ComboBox control displays state names that use a two-letter format, or the entire state name.

Defining properties by using setters and getters

You can define properties for your MXML components by using setter and getter methods. The advantage of getters and setters is that they isolate the variable from direct public access so that you can perform the following tasks:

- Inspect and validate any data written to the property on a write operation
- Trigger events that are associated with the property when the property changes
- Calculate a return value on a read operation

For more information, see "Custom ActionScript Components" on page 13.

In the following example, the StateComboBoxGetSet.mxml component contains several new properties and methods:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxSetGet.mxml -->

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

    <mx:Script>
        <![CDATA[

            // Define private variables.
            private var stateArrayShort:Array = ["AK", "AL"];
            private var stateArrayLong:Array = ["Arkansas", "Alaska"];

            // Variable holding the display setting.
            private var __shortNames:Boolean = true;
            // Set method.
            public function set shortNames(val:Boolean):void {
                // Call method to set the dataProvider
                // based on the name length.
                __shortNames = val;
                if (__shortNames) {
                    this.dataProvider=stateArrayShort; }
                else {
                    this.dataProvider=stateArrayLong; }
            }

            // Get method.
            public function get shortNames():Boolean{
                return __shortNames;
            }
        ]]>
    </mx:Script>
</mx:ComboBox>
```

In this example, you create a StateComboBoxGetSet.mxml control that takes a `shortNames` property defined by using ActionScript setter and getter methods. One advantage to using setter and getter methods to define a property is that the component can recognize changes to the property at run time. For example, you can give your users the option of displaying short state names or long state names from the application. The setter method modifies the component at run time in response to the user's selection.

You can also define events to be dispatched when a property changes. This enables you to signal the change so that an event listener can recognize the change. For more information on events, see "Working with events" on page 93.

You can call a component's custom methods and access its properties in ActionScript just as you would any instance method or component property, as the following application shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropSetGet.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateComboBoxSetGet id="myStateCB" shortNames="true"/>
    <mx:Button click="myStateCB.shortNames=!myStateCB.shortNames;"/>

</mx:Application>
```

In this example, selecting the button toggles the display format of the state name between the short and long formats.

Defining inspectable properties

You should precede the variable or set function with the `[Inspectable]` metadata tag if you plan to use the component in an authoring tool such as Adobe Flex Builder™. The `[Inspectable]` metadata tag must immediately precede the property's variable declaration or the setter and getter methods to be bound to that property, as the following example shows:

```
<mx:Script>
    <![CDATA[

        // Define public variables.
        [Inspectable(defaultValue=true)]
        public var shortNames:Boolean = true;

    ]]>
</mx:Script>
```

For more information on the `[Inspectable]` metadata tag, see "Metadata Tags in Custom Components" on page 33.

Supporting data binding in custom properties

The Flex data binding mechanism provides a syntax for automatically copying the value of a property of one object to a property of another object at run time. The following example shows a Text control that gets its data from 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`, to the destination property, the Text control's `text` property, as the following example shows:

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

Data binding is usually triggered whenever the value of the source property changes.

Properties that you define in your custom controls can also take advantage of data binding. You can automatically use any property defined by using an MXML tag, such as `<mx:Boolean>`, and any ActionScript property defined as a variable or defined by using setter and getter methods as the destination of a binding expression.

For example, "Defining properties by using setters and getters" on page 84 defined the `shortNames` property of StateComboBoxGetSet.mxml by using setter and getter methods. With no modification to that component, you can use `shortNames` as the destination of a binding expression, as the following example shows:

```
<MyComp:StateComboBoxSetGet shortNames="{some_prop}"/>
```

However, you can also write your component to use the `shortNames` property as the source of a binding expression, as the following example shows for the component StateComboBoxGetSetBinding.mxml:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPropSetGetBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateComboBoxSetGetBinding id="myStateCB" shortNames="false"/>

    <mx:TextArea text="The value of shortNames is {myStateCB.shortNames}"/>

    <mx:Button click="myStateCB.shortNames=!myStateCB.shortNames;"/>

</mx:Application>
```

When a property is the source of a data binding expression, any changes to the property must signal an update to the destination property. The way to signal that change is to dispatch an event, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxSetGetBinding.mxml -->

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

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private var stateArrayShort:Array = ["AK", "AL"];
            private var stateArrayLong:Array = ["Arkansas", "Alaska"];

            private var __shortNames:Boolean = true;

            public function set shortNames(val:Boolean):void {
                __shortNames = val;
                if (__shortNames) {
                    dataProvider=stateArrayShort; }
                else {
                    dataProvider=stateArrayLong; }
                    // Create and dispatch event.
                    dispatchEvent(new Event("changeShortNames"));
            }

            // Include the [Bindable] metadata tag.
            [Bindable(event="changeShortNames")]
            public function get shortNames():Boolean {
                return __shortNames;
            }
        ]]>
    </mx:Script>
</mx:ComboBox>
```

Use a property as the source of a data binding expression

1 Define the property as a variable, or by using setter and getter methods.

You must define a setter method and a getter method if you use the `[Bindable]` tag with the property.

2 Insert the `[Bindable]` metadata tag before the property definition, or before either the setter or getter method, and optionally specify the name of the event dispatched by the property when it changes.

If you omit the event name specification from the `[Bindable]` metadata tag, Flex automatically generates and dispatches an event named `propertyChange`. If the property value remains the same on a write, Flex does not dispatch the event or update the property.

Alternatively, you can place the `[Bindable]` metadata before a public class definition. This makes all public properties that you defined as variables, and all public properties that you defined by using both a setter and a getter method, usable as the source of a binding expression.

***Note:** When you use the `[Bindable]` metadata tag before a public class definition, it only applies to public properties; it does not apply to private or protected properties, or to properties defined in any other namespace. You must insert the `[Bindable]` metadata tag before a nonpublic property to make it usable as the source for a data binding expression.*

3 Add a call to the `dispatchEvent()` method to dispatch the event when you define the event name in the `[Bindable]` metadata tag.

For more information on using the `[Bindable]` tag, see "Bindable metadata tag" on page 38.

Passing references to properties of MXML components

One of the ways that you can make a component reusable is to design it so that users can pass values to the component by using public properties of the component. For information on how to define properties for MXML components by using MXML and ActionScript, and how to pass values to those properties, see "Supporting data binding in custom properties" on page 86.

Rather than passing a value to a component, you can pass a reference to it. The reference could be to the calling component, to another component, or to a property of a component.

Accessing the Application object

The Application object is the top-level object in a Flex application. Often, you must reference properties or objects of the Application object from your custom component. Use the `mx.core.Application.application` static property to reference the application object.

You can also use the `parentDocument` property to reference the next object up in the document chain of a Flex application. The `parentDocument` property is inherited by all components from the UIComponent class. For an MXML component, the `parentDocument` property references the Object corresponding to the component that referenced the MXML component.

For more information on the `mx.core.Application.application` static property and the `parentDocument` property, see "Application Container" on page 451 in *Adobe Flex 3 Developer Guide*.

Even if the calling file does not pass a reference to the Application object, you can always access it from your MXML component. For example, the following application contains a custom component called StateComboBoxDirectRef. In this example, StateComboBoxDirectRef is designed to write the index of the selected item in the ComboBox to the TextArea control:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainDirectRef.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:TextArea id="myTAMain"/>
    <MyComp:StateComboBoxDirectRef/>

</mx:Application>
```

The simplest way to write StateComboBoxDirectRef.mxml is to use the `mx.core.Application.application` static property to write the index directly to the TextArea control, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxDirectRef.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    close="handleCloseEvent(event);">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;
            import mx.core.Application;

            public function handleCloseEvent(eventObj:Event):void {
                mx.core.Application.application.myTAMain.text=
                    String(this.selectedIndex);
            }
        ]]>
    </mx:Script>

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

In the previous example, you use the `close` event of the ComboBox control to write the `selectedIndex` directly to the TextArea control in the main application. You must cast the value of `selectedIndex` to a String because the `text` property of the TextArea control is of type String.

You could make the custom component slightly more reusable by using the `parentDocument` property to reference the TextArea control, rather than the `mx.core.Application.application` static property. By using the `parentDocument` property, you can call the custom component from any other MXML component that contains a TextArea control named myTAMain, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxDirectRefParentObj.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    close="handleCloseEvent(event);">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            public function handleCloseEvent(eventObj:Event):void {
                parentDocument.myTAMain.text=String(selectedIndex);
            }
        ]]>
    </mx:Script>

    <mx:dataProvider>
        <mx:Array>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
        </mx:Array>
    </mx:dataProvider>
</mx:ComboBox>
```

Although these examples work, they require that the TextArea control has a predefined `id` property, and that MXML component knows that `id`. In this case, the custom component is an example of a tightly coupled component. That is, the component is written for a specific application and application structure, and it is not easily reused in another application.

Passing a reference to the component

A loosely coupled component is a highly reusable component that you can easily use in different places in one application, or in different applications. To make the component from "Supporting data binding in custom properties" on page 86 reusable, you can pass a reference to the TextArea control to the custom component, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainPassRefToTA.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:TextArea id="myTAMain" />
    <MyComp:StateComboBoxPassRefToTA outputTA="{myTAMain}" />

</mx:Application>
```

The custom component does not have to know anything about the main application, other than that it writes its results back to a TextArea control, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/StateComboBoxPassRefToTA.mxml -->

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml" close="handleCloseEvent(event);">

    <mx:Script>
        <![CDATA[
            import flash.events.Event;
            import mx.controls.TextArea;

            // Define a variable of type mx.controls.TextArea.
            public var outputTA:TextArea;

            public function handleCloseEvent(eventObj:Event):void {
                outputTA.text=String(this.selectedIndex);
            }
        ]]>
    </mx:Script>

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

In this example, you use the Flex data binding syntax to pass the reference to the TextArea control to your custom component. Now, you can use StateComboBoxPassRefToTA.mxml anywhere in an application. The only requirement is that the calling component must pass a reference to a TextArea control to the component.

Passing a reference to the calling component

In “Passing a reference to the component” on page 91, you passed a reference to a single component to the custom MXML component. This allowed the MXML component to access only a single component in the main application.

One type of reference that you can pass to your component is a reference to the calling component. With a reference to the calling component, your custom MXML file can access any properties or object in the calling component.

To pass a reference to the calling component to a custom MXML component, you create a property in the custom MXML component to represent the calling component. Then, from the calling component, you pass the reference, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/CallingComponent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="left"
    xmlns:MyComp="myComponents.*">

    <!-- Use the caller property to pass a reference to the
        calling component to DestinationComp. -->

    <mx:Label text="Enter text"/>
    <mx:TextInput id="text1" text="Hello"/>

    <mx:Label text="Input text automatically copied to MXML component."/>
    <MyComp:DestinationComp caller="{this}"/>

</mx:Application>
```

In the definition of DestinationComp.mxml, you define the `caller` property, and specify as its data type the name of the file of the calling MXML file, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/DestinationComp.mxml -->

<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            // Define variable to reference calling file.
            [Bindable]
            public var caller:CallingComponent;
        ]]>
    </mx:Script>

    <mx:TextInput id="mytext" text="{caller.text1.text}"/>
</mx:VBox>
```

Remember, an MXML component corresponds to an ActionScript class, where the ActionScript class name is the filename of the MXML component. Therefore, the MXML component defines a new data type. You can then create a variable whose data type is that of the calling file.

With the reference to the calling file, your MXML component can access any property of the calling file, and you can bind the value of the TextInput control in CallingComp.mxml to the TextInput control in StateComboBox.mxml. Creating a property of type CallingComp provides strong typing benefits and ensures that binding works correctly.

Working with events

Flex applications are event-driven. Events let a programmer know when the user interacts with the interface, and also when important changes happen in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing. You can handle events that your custom components generate and add your own event types to your custom components.

Handling events from simple MXML components

Simple MXML components are those that contain a single root tag that is not a container. In this topic, the StateComboBox.mxml component is a simple component because it contains a definition only for the ComboBox control.

You have two choices for handling events that a simple component dispatches: handle the events within the definition of your MXML component, or allow the file that references the component to handle them.

The following example uses the StateComboBox.mxml component, and defines the event listener for the component's `close` event in the main application:

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

    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            public function handleCloseEvent(eventObj:Event):void {
                ...
            }
        ]]>
    </mx:Script>

    <MyComp:StateComboBox rowCount="5" close="handleCloseEvent(event);"/>

</mx:Application>
```

In this example, if the MXML component dispatches a `close` event, the event listener in the calling MXML file handles it.

Alternatively, you could define the event listener within the StateComboBox.mxml component, as the following example shows:

```
<?xml version="1.0"?>
<!-- StateComboBox.mxml -->
<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml"
    close="handleCloseEvent(event);">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            public function handleCloseEvent(eventObj:Event):void {
                ...
            }
        ]]>
    </mx:Script>

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

With simple MXML components, you can define event listeners in both places, and both event listeners process the event. However, the event listeners defined within the component execute before any listeners defined in the application.

Creating custom events

All MXML components can dispatch events, either those inherited by the components from their superclasses, or new events that you define within your components. When you are developing MXML components, you can add your own event types.

In this example, you define a new component called TextAreaEnabled.mxml that uses a `<mx:TextArea>` tag as its root tag. This component also defines a new property called `enableTA` that users set to `true` to enable text input or to `false` to disable input.

The setter method dispatches a new event type, called `enableChanged`, when the value of the `enableTA` variable changes. The `[Event]` metadata tag identifies the event to the MXML compiler so that the file referencing the component can use the new property. For more information on using the `[Event]` metadata keyword, see "Metadata Tags in Custom Components" on page 33.

The syntax for the `[Event]` metadata tag is as follows:

```
<mx:Metadata>
    [Event(name="eventName", type="eventType")]
</mx:Metadata>
```

You dispatch new event types by using the `dispatchEvent()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/TextAreaEnabled.mxml -->

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

    <mx:Metadata>
        [Event(name="enableChanged", type="flash.events.Event")]
    </mx:Metadata>

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            // Define private variable to hold the enabled state.
            private var __enableTA:Boolean;

            // Define a setter method for the private variable.
            public function set enableTA(val:Boolean):void {
                __enableTA = val;
                enabled = val;

                // Define event object, initialize it, then dispatch it.
                dispatchEvent(new Event("enableChanged"));
            }

            // Define a getter method for the private variable.
            public function get enableTA():Boolean {
               return __enableTA;
            }
        ]]>
    </mx:Script>
</mx:TextArea>
```

The following main application includes TextAreaEnabled.mxml and defines an event listener for the `enableChanged` event:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainTextAreaEnable.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;
            import myComponents.TextAreaEnabled;

            public function handleEnableChangeEvent(eventObj:Event):void {
                var tempTA:TextAreaEnabled =
                    eventObj.currentTarget as TextAreaEnabled;
                if (tempTA.enableTA) {
                    myButton.label="Click to disable";
                    }
                else {
                    myButton.label="Click to enable";
                    }
            }
        ]]>
    </mx:Script>

    <MyComp:TextAreaEnabled id="myTA" enableTA="false"
        enableChanged="handleEnableChangeEvent(event);" />

    <mx:Button id="myButton" label="Click to enable"
        click="myTA.enableTA=!myTA.enableTA;" />

</mx:Application>
```

If you do not use the `[Event]` metadata tag in the custom component file to define the `enableChanged` event, the MXML compiler generates an error message when you reference the event name in an MXML file. Any component can register an event listener for the event in ActionScript using the `addEventListener()` method, even if you omit the `[Event]` metadata tag.

You can also create and dispatch events that use an event object of a type other than that defined by the Event class. For example, you might want to create an event object that contains new properties so that you can pass those properties back to the referencing file. To do so, you create a subclass of the Event class to define your new event object. For information on creating custom event classes, see "Custom Events" on page 23.

Handling events from composite components

Composite components are components that use a container for the root tag, and define child components in that container. You handle events generated by the root container in the same way as you handle events generated by simple MXML components. That is, you can handle the event within the MXML component, within the referencing file, or both. For more information, see "Handling events from simple MXML components" on page 93.

To handle an event that a child of the root container dispatches, you can handle it in the MXML component in the same way as you handle an event from the root container. However, if a child component of the root container dispatches an event, and you want that event to be dispatched to the referencing file, you must add logic to your custom component to propagate the event.

For example, you can define a component that uses an `<mx:Form>` tag as the root tag, and include within it a ComboBox control. Any event that the Form container dispatches, such a `scroll` event, is dispatched to the referencing file of the custom component. However, the `close` event of the ComboBox control is dispatched only within the custom MXML component.

To propagate the `close` event outside of the custom component, you define an event listener for it in the MXML component that redispatches it, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/AddressForm.mxml -->

<mx:Form xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:local="*">

    <mx:Metadata>
        [Event(name="close", type="flash.events.Event")]
    </mx:Metadata>

    <mx:Script>
        <![CDATA[

            import flash.events.Event;
            // Redispatch event.
            private function handleCloseEventInternal(eventObj:Event):void {
                dispatchEvent(eventObj);
            }
        ]]>
    </mx:Script>

    <mx:FormItem label="Name">
        <mx:TextInput id="name1" />
    </mx:FormItem>

    <mx:FormItem label="Street">
        <mx:TextInput id="street" />
    </mx:FormItem>

    <mx:FormItem label="City" >
        <mx:TextInput id="city" />
    </mx:FormItem>

    <mx:FormItem label="State" >
        <mx:ComboBox close="handleCloseEventInternal(event);">
            <mx:dataProvider>
                <mx:Array>
                    <mx:String>AK</mx:String>
                    <mx:String>AL</mx:String>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
    </mx:FormItem>
</mx:Form>
```

In this example, you propagate the event to the calling file. You could, alternatively, create an event type and new event object as part the propagation. For more information on the `[Event]` metadata tag, see “Metadata Tags in Custom Components” on page 33.

You can handle the `close` event in your main application, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainAddressFormHandleEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <mx:Script>
        <![CDATA[

            import flash.events.Event;
            private function handleCloseEvent(eventObj:Event):void {
                myTAClose.text=eventObj.type;
            }

            private function handleMouseDown(eventObj:Event):void {
                myTA.text=eventObj.type;
            }
        ]]>
    </mx:Script>

    <mx:TextArea id="myTA" />
    <mx:TextArea id="myTAClose" />

    <MyComp:AddressForm mouseDown="handleMouseDown(event);"
        close="handleCloseEvent(event);"/>
</mx:Application>
```

About interfaces

Interfaces are a type of class that you design to act as an outline for your components. When you write an interface, you provide only the names of public methods rather than any implementation. For example, if you define two methods in an interface and then implement that interface, the implementing class must provide implementations of those two methods.

Interfaces in ActionScript can declare methods and properties only by using setter and getter methods; they cannot specify constants. The benefit of interfaces is that you can define a contract that all classes that implement that interface must follow. Also, if your class implements an interface, instances of that class can also be cast to that interface.

Custom MXML components can implement interfaces just as other ActionScript classes can. To do this, you use the `implements` attribute. All MXML tags support this attribute.

The following code is an example of a simple interface that declares several methods:

```
// The following is in a file named SuperBox.as.
interface SuperBox {
    function selectSuperItem():String;
    function removeSuperItem():Boolean;
    function addSuperItem():Boolean;
}
```

A class that implements the SuperBox interface uses the `implements` attribute to point to its interface and must provide an implementation of the methods. The following example of a custom ComboBox component implements the SuperBox interface:

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

<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml" implements="SuperBox">
    <mx:Script>
        <![CDATA[
            public function selectSuperItem():String {
                return "Super Item was selected";
            }
            public function removeSuperItem():Boolean {
                return true;
            }
            public function addSuperItem():Boolean {
                return true;
            }
        ]]>
    </mx:Script>
    <mx:dataProvider>
        <mx:String>AK</mx:String>
        <mx:String>AL</mx:String>
    </mx:dataProvider>
</mx:ComboBox>
```

You can implement multiple interfaces by separating them with commas, as the following example shows:

```
<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml" implements="SuperBox, SuperBorder,
SuperData">
```

All methods that you declare in an interface are considered public. If you define an interface and then implement that interface, but do not implement all of its methods, the MXML compiler throws an error.

Methods that are implemented in the custom component must have the same return type as their corresponding methods in the interface. If no return type is specified in the interface, the implementing methods can declare any return type.

About implementing IMXMLObject

You cannot define a constructor for an MXML component. If you do, the Flex compiler issues an error message that specifies that you defined a duplicate function.

For many types of Flex components, you can use an event listener instead of a constructor. For example, depending on what you want to do, you can write an event listener for the `preinitialize`, `initialize`, or `creationComplete` event to replace the constructor.

These events are all defined by the UIComponent class, and inherited by all of its subclasses. If you create an MXML component that is not a subclass of UIComponent, you cannot take advantage of these events. You can instead implement the IMXMLObject interface in your MXML component, and then implement the `IMXMLObject.initialized()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/myComponents/ObjectComp.mxml -->

<mx:Object xmlns:mx="http://www.adobe.com/2006/mxml"
    implements="mx.core.IMXMLObject">

    <mx:Script>
        <![CDATA[

            // Implement the IMXMLObject.initialized() method.
            public function initialized(document:Object, id:String):void {
                trace("initialized, x = " + x);
            }
        ]]>
    </mx:Script>

    <mx:Number id="y"/>
    <mx:Number id="z"/>
    <mx:Number id="x"/>
</mx:Object>
```

Flex calls the `IMXMLObject.initialized()` method after it initializes the properties of the component. The following example uses this component:

```
<?xml version="1.0"?>
<!-- mxmlAdvanced/MainInitObject.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[

            public function initApp():void {
                myTA.text="myFC.x = " + String(myFC.x);
            }
        ]]>
    </mx:Script>
    <MyComp:ObjectComp id="myFC" x="1" y="2" z="3"/>
    <mx:TextArea id="myTA"/>

</mx:Application>
```

Because Flex calls the `IMXMLObject.initialized()` method after it initializes the properties of the component, the `trace()` function in the implementation of the `IMXMLObject.initialized()` method outputs the following:

```
initialized, x = 1
```

Part 3: ActionScript Custom Components

Topics

Chapter 8: Simple Visual Components in ActionScript

You define custom ActionScript components to extend the Adobe® Flex® component library. For example, you can create a customized Button, Tree, or DataGrid component as an ActionScript component.

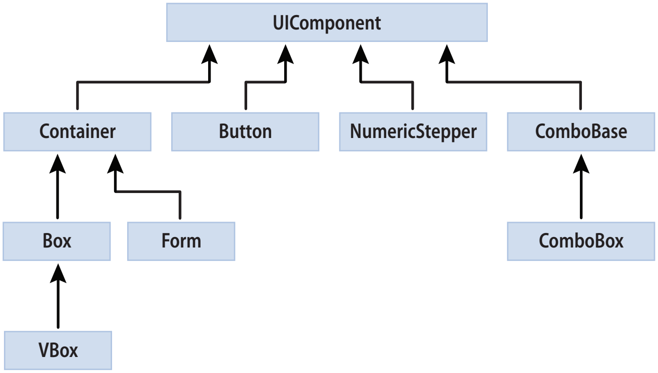
For information on creating advanced components in ActionScript, see "Advanced Visual Components in ActionScript" on page 129.

Topics

About ActionScript components

You create reusable components by using 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.

Flex components are implemented as a class hierarchy in ActionScript. Each component in your application is an instance of an ActionScript class. The following example shows just a portion of this hierarchy:



Note: *This example shows a portion of the class hierarchy. For a complete description of the class hierarchy, see the Adobe Flex Language Reference.*

All Flex visual components are derived from the ActionScript UIComponent class. To create your own components, you can create a subclass from the UIComponent class or from any of its subclasses.

The class you choose to use as the superclass of your custom component depends on what you are trying to accomplish. For example, you might require a custom button control. You could create a subclass of the UIComponent class, and then recreate all of the functionality built into the Flex Button class. A better and faster way to create your custom button component is to create a subclass of the Flex Button class, and then modify it in your custom class.

Simple components are subclasses of existing Flex components that modify the behavior of the component, or add new functionality to it. For example, you might add a new event type to a Button control, or modify the default styles or skins of a DataGrid control.

You can also create advanced ActionScript components. Advanced ActionScript components might have one of the following requirements:

- Modify the appearance of a control or the layout functionality of a container
- Encapsulate one or more components into a composite component
- Subclass UIComponent to create components

For information on creating advanced ActionScript components, see "Advanced Visual Components in ActionScript" on page 129.

Example: Creating a simple component

When you define a simple component, you do not create a component yourself, but instead modify the behavior of an existing component. In this section, you create a customized TextArea control by extending the mx.controls.TextArea component. This component adds an event listener for the `keyDown` event to the TextArea control. The `KeyDown` event deletes all the text in the control when a user presses the Control+Z key combination.

```
package myComponents
{
    // as/myComponents/DeleteTextArea.as
    import mx.controls.TextArea;
    import flash.events.KeyboardEvent;

    public class DeleteTextArea extends TextArea {

        // Constructor
        public function DeleteTextArea() {
            // Call super().
            super();

            // Add event listener for keyDown event.
            addEventListener("keyDown", myKeyDown);
        }

        // Define private keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Check to see if Ctrl-Z pressed. Keycode for Z is 90.
            if (eventObj.ctrlKey && eventObj.keyCode == 90)
                text = "";
        }
    }
}
```

The filename for this component is DeleteTextArea.as, and its location is the myComponents subdirectory of the application, as specified by the `package` statement. For more information on using the `package` statement and specifying the directory location of your components, see “Custom ActionScript Components” on page 13.

You can now use your new TextArea control in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/MainDeleteTextArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:DeleteTextArea/>

</mx:Application>
```

***Note:** Your class must be specified as `public` for you to be able to access it by using an MXML tag.*

In this example, you first define the `MyComp` namespace to specify the location of your custom component. You then reference the component as an MXML tag by using the namespace prefix.

You can specify any inherited properties of the superclass in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/MainDeleteTextAreaProps.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:DeleteTextArea wordWrap="true" text="My Message"/>

</mx:Application>
```

You do not have to change the name of your custom component when you create a subclass of a Flex class. In the previous example, you could have named your custom component TextArea, and written it to the TextArea.as file in the myComponents directory, as the following example shows:

```
package myComponents
{
    import mx.controls.TextArea;
    import flash.events.KeyboardEvent;

    public class TextArea extends mx.controls.TextArea {
        ...
    }
}
```

You can now use your custom TextArea control, and the standard TextArea control, in an application. To differentiate between the two controls, you use the namespace prefix, as the following example shows:

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

Adding properties and methods to a component

To make your custom components reusable, you design them so that users can pass information to them. You do this by adding public properties and methods to your components, and by making the components accessible in MXML.

Defining public properties in ActionScript

You can use one of the following methods to add public properties to your ActionScript components:

- Define public variables
- Define public getter and setter methods

Accessing public properties in MXML

All public properties defined in your component are accessible in MXML by using MXML tag properties. For example, you might allow the user to pass a value to your component, as the following example shows:

```
<MyComp:MyCustomComponent prop1="3"/>
```

To create a component that takes tag attributes in MXML, you define a public variable with the same name as the tag attribute in your class definition:

```
public class MyCustomComponent extends TextArea {

    // Define an uninitialized variable.
    public var prop1:Number;

    // Define and initialize a variable.
    public var prop2:Number=5;
    ...
}
```

You can also use public getter and setter methods to define a property, as the following example shows:

```
public class MyCustomComponent extends TextArea {

    private var _prop1:Number;

    public function get prop1():Number {
        // Method body.
        // Typically the last line returns the value of the private variable.
        return _prop1;
    }

    public function set prop1(value:Number):void {
        // Typically sets the private variable to the argument.
        _prop1=value;
        // Define any other logic, such as dispatching an event.
    }

}
```

You can define and initialize a private variable, as the following example shows:

```
private var _prop2:Number=5;
```

When you specify a value to the property in MXML, Flex automatically calls the setter method. If you do not set the property in MXML, Flex sets it to its initial value, if you specified one, or to the type's default value, which is `NaN` for a variable of type Number.

Defining public properties as variables

In the following example, you use the Control+I key combination to extend the TextArea control to let the user increase the font size by one point, or use the Control+M key combination to decrease the font size by one point:

```
package myComponents
{

    // as/myComponents/TextAreaFontControl.as
    import mx.controls.TextArea;
    import flash.events.KeyboardEvent;
    import flash.events.Event;

    public class TextAreaFontControl extends TextArea
    {
        // Constructor
        public function TextAreaFontControl() {
            super();

            // Add event listeners.
            addEventListener("keyDown", myKeyDown);
            addEventListener("creationComplete", myCreationComplete);
        }

        // Define private var for current font size.
        private var currentFontSize:Number;

        // Define a public property for the minimum font size.
        public var minFontSize:Number = 5;
        // Define a public property for the maximum font size.
        public var maxFontSize:Number = 15;

        // Initialization event handler for getting default font size.
        private function myCreationComplete(eventObj:Event):void {
            // Get current font size
             currentFontSize = getStyle('fontSize');
        }

        // keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Was Ctrl key pressed?
            if (eventObj.ctrlKey)
            {
                switch (eventObj.keyCode) {
                    // Was Ctrl-I pressed?
                    case 73 :
                        if (currentFontSize < maxFontSize) {
                            currentFontSize = currentFontSize + 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    // Was Ctrl-M pressed?
                    case 77 :
                        if (currentFontSize > minFontSize) {
                            currentFontSize = currentFontSize - 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    default :
                        break;
                }
            }
        }
    }
}
```

Notice that the call to the `getStyle()` method is in the event listener for the `creationComplete` event. You must wait until component creation is complete before calling `getStyle()` to ensure that Flex has set all inherited styles. However, you can call `setStyle()` in the component constructor to set styles.

This example uses variables to define public properties to control the maximum font size, `maxFontSize`, and minimum font size, `minFontSize`, of the control. Users can set these properties in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/MainTextAreaFontControl.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:TextAreaFontControl id="myTAFS"
        minFontSize="8"
        maxFontSize="50"/>

    <mx:Button
        label="Get Font Size"
        click="myTA.text=String(myTAFS.getStyle('fontSize'));"/>
    <mx:TextArea id="myTA"/>
</mx:Application>
```

Defining public properties by using getter and setter methods

There are no restrictions on using public variables to define public properties, However, Adobe recommends that you use getter and setter methods so that you can control user interaction with your component, as described in "Defining properties as getters and setters" on page 18.

The following example code defines a component named TextAreaFontControlGetSet that replaces the public property definition for the `maxFontSize` property shown in "Defining public properties as variables" on page 110:

```
package myComponents
{
    // as/myComponents/TextAreaFontControlGetSet.as
    import mx.controls.TextArea;
    import flash.events.KeyboardEvent;
    import flash.events.Event;

    public class TextAreaFontControlGetSet extends TextArea
    {
        public function TextAreaFontControlGetSet()
        {
            super();
            addEventListener("keyDown", myKeyDown);
            addEventListener("creationComplete", myCreationComplete);
        }

        private var currentFontSize:Number;
        public var minFontSize:Number = 5;

        // Define private variable for maxFontSize.
        private var _maxFontSize:Number = 15;

        // Define public getter method.
        public function get maxFontSize():Number {
            return _maxFontSize;
        }

        // Define public setter method.
        public function set maxFontSize(value:Number):void {
            if (value <= 30) {
                _maxFontSize = value;
            } else _maxFontSize = 30;
        }
        private function myCreationComplete(eventObj:Event):void {
            // Get current font size
             currentFontSize = getStyle('fontSize');
        }

        // keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Was Ctrl key pressed?
            if (eventObj.ctrlKey)
            {
                switch (eventObj.keyCode) {
                    // Was Ctrl-I pressed?
                    case 73 :
                        if (currentFontSize < maxFontSize) {
                            currentFontSize = currentFontSize + 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    // Was Ctrl-M pressed?
                    case 77 :
                        if (currentFontSize > minFontSize) {
                            currentFontSize = currentFontSize - 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    default :
                        break;
                }
            }
        }
    }
}
```

In this example, the setter method checks that the specified font size is less than the predefined limit of 30 pixels. If the font size is greater than the limit, it sets it to the limit.

Creating a default property

You can define a default property for your ActionScript components by using the `[DefaultProperty]` metadata tag. You can then use the default property in MXML as the child tag of the component tag without specifying the property name. For more information on using the default property, including an example, see “Defining public properties as variables” on page 110 in *Adobe Flex 3 Developer Guide*.

You can use the `[DefaultProperty]` metadata tag in your ActionScript component to define a single default property, as the following example shows:

```
package myComponents
{
    // as/myComponents/TextAreaDefaultProp.as
    import mx.controls.TextArea;

    // Define the default property.
    [DefaultProperty("defaultText")]

    public class TextAreaDefaultProp extends TextArea {

        public function TextAreaDefaultProp()
        {
            super();
        }

        // Define a setter method to set the text property
        // to the value of the default property.
        public function set defaultText(value:String):void {
            if (value!=null)
            text=value;
        }

        public function get defaultText():String {
            return text;
        }
    }
}
```

In this example, you add a new property to the TextArea control, called `defaultProperty`, and specify it as the default property of the control. The setter method for `defaultProperty` just sets the value of the `text` property of the control. You can then use the default property in MXML, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/MainTextAreaDefaultProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:TextAreaDefaultProp>Hello</MyComp:TextAreaDefaultProp>

</mx:Application>
```

The one place where Flex prohibits the use of a default property is when you use the ActionScript class as the root tag of an MXML component. In this situation, you must use child tags to define the property, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/myComponents/TextAreaDefaultPropMXML.mxml -->
```

```
<MyComp:TextAreaDefaultProp xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

        <MyComp:defaultText>Hello</MyComp:defaultText>

</MyComp:TextAreaDefaultProp>
```

Making properties accessible in Flex Builder

You can make your property definitions accessible in Adobe Flex Builder by adding the `[Inspectable]` metadata tag to the property definition. For example, if you are using Flex Builder, you can insert the `[Inspectable]` metadata tag to define the property as user-editable (or *inspectable*), as the following example shows:

```
[Inspectable]
var prop1:Number;
```

You can also use the `[Inspectable]` metadata tag with setter and getter methods. For more information, see "Metadata Tags in Custom Components" on page 33.

Using data binding with custom properties

Data binding defines a syntax for automatically copying the value of a property of one object, the *source* property, to a property of another object, the *destination* property, at run time. Data binding is usually triggered when the value of the source property changes.

The following example shows a Text control that gets its data from a HSlider control's `value` property. The property name inside the curly braces ({ }) specifies a binding expression that copies the value of the source property, `mySlider.value`, into the destination property, the Text control's `text` property.

```
<mx:HSlider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

The current value of the HSlider control appears in the Text control when you stop moving the slider. To get continuous updates as you move the slider, set the `HSlider.liveDragging` property to `true`.

Using properties as the destination of a binding expression

Properties in your custom components can take advantage of data binding. Any property defined as a variable or defined by using a setter and getter method can automatically be used as the destination of a binding expression.

For example, in the section "Defining public properties in ActionScript" on page 109, you created a class with the public property `maxFontSize`. You can use the `maxFontSize` property as the destination of a binding expression, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/MainTextAreaFontControlBindingDest.mxml -->
```

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

    <MyComp:TextAreaFontControl id="myTA"
        maxFontSize="{Number(myTI.text)}"/>

    <mx:Label text="Enter max font size."/>
    <mx:TextInput id="myTI" text="25"/>

</mx:Application>
```

In this example, any value that the user enters into the TextInput control is automatically copied to the `maxFontSize` property.

Using properties as the source of a data binding expression

When a property is the source of a data binding expression, Flex automatically copies the value of the source property to any destination property when the source property changes. However, in order to signal to Flex to perform the copy, you must register the property with Flex and the source property must dispatch an event.

To register a property as a source for data bindings, you use the `[Bindable]` metadata tag. You can use this tag in three places:

- Before a class definition, in order to make all public properties defined as variables usable as the source of a binding expression. You can also do this by using setter and getter methods, in which case you use the `[Bindable]` tag before the getter method.
- Before a property that a variable defines, in order to make that specific property support binding
- Before a getter method for a property implemented by using setter and getter methods

***Note:** When you use the `[Bindable]` metadata tag before a public class definition, it applies only to public properties; it does not apply to private or protected properties, or to properties defined in any other namespace. You must insert the `[Bindable]` metadata tag before a nonpublic property to make it usable as the source for a data binding expression.*

For more information on the `[Bindable]` metadata tag, see “Metadata Tags in Custom Components” on page 33.

The following example modifies the component in the section “Defining public properties in ActionScript” on page 109 to make the `maxFontSize` and `minFontSize` properties usable as the source for data bindings:

```
    // Define public properties for tracking font size.
    [Bindable]
    public var maxFontSize:Number = 15;
    [Bindable]
    public var minFontSize:Number = 5;
```

If you omit the event name from the `[Bindable]` metadata tag, Flex automatically dispatches an event named `propertyChange` when the property changes to trigger the data binding. If the property value remains the same on a write operation, Flex does not dispatch the event or update the property.

When you define a property by using getter and setter methods so that the property is usable as the source for data binding, you include the `[Bindable]` metadata tag before the getter method, and optionally include the name of the event dispatched by the setter method when the property changes, as the following example shows:

```
package myComponents
{
    // as/myComponents/TextAreaFontControlBinding.as
    import mx.controls.TextArea;
    import flash.events.KeyboardEvent;
    import flash.events.Event;

    public class TextAreaFontControlBinding extends TextArea
    {
        public function TextAreaFontControlBinding()
        {
            super();
            addEventListener("keyDown", myKeyDown);
            addEventListener("creationComplete", myCreationComplete);
        }

        private var currentFontSize:Number;
        public var minFontSize:Number = 5;

        // Define private variable for maxFontSize.
        public var _maxFontSize:Number = 15;

        // Define public getter method, mark the property
        // as usable for the source of data binding,
        // and specify the name of the binding event.
        [Bindable("maxFontSizeChanged")]
        public function get maxFontSize():Number {
            return _maxFontSize;
        }

        // Define public setter method.
        public function set maxFontSize(value:Number):void {
            if (value <= 30) {
                _maxFontSize = value;
            } else _maxFontSize = 30;

            // Dispatch the event to trigger data binding.
            dispatchEvent(new Event("maxFontSizeChanged"));
        }

        private function myCreationComplete(eventObj:Event):void {
            // Get current font size
             currentFontSize = getStyle('fontSize');
        }

        // keyDown event handler.
        private function myKeyDown(eventObj:KeyboardEvent):void {
            // Was Ctrl key pressed?
            if (eventObj.ctrlKey)
            {
                switch (eventObj.keyCode) {
                    // Was Ctrl-I pressed?
                    case 73 :
                        if (currentFontSize < maxFontSize) {
                            currentFontSize = currentFontSize + 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    // Was Ctrl-M pressed?
                    case 77 :
                        if (currentFontSize > minFontSize) {
                            currentFontSize = currentFontSize - 1;
                            setStyle('fontSize', currentFontSize);
                        }
                        break;
                    default :
                        break;
                }
            }
        }

    }
}
```

In this example, the setter updates the value of the property, and then dispatches an event to trigger an update of any data binding destination. The name of the event is not restricted. You can use this component in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- as/MainTextAreaFontControlBindingSource.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:TextAreaFontControlBinding id="myTA"
        maxFontSize="{Number(myTI.text)}"/>
    <mx:Label text="Enter max font size."/>
    <mx:TextInput id="myTI" text="15"/>

    <mx:Label text="Current max font size."/>
    <mx:TextArea text="{String(myTA.maxFontSize)}"/>

</mx:Application>
```

Defining a method override

You can override a method of a base class in your ActionScript component. To override the method, you add a method with the same signature to your class, and prefix it with the `override` keyword. The following example overrides the `HBox.addChild()` method to open an Alert box when a new item is added to it:

```
package myComponents
{
    import mx.controls.Alert;
    import mx.containers.HBox;
    import flash.display.DisplayObject;

    public class HBoxWithAlert extends HBox
    {
        // Define the constructor.
        public function HBoxWithAlert()
        {
            super();
        }

      // Define the override.
      override public function addChild(child:DisplayObject):DisplayObject {

            // Call super.addChild().
            super.addChild(child);

            // Open the Alert box.
            Alert.show("Item added successfully");

            return child;
        }
    }
}
```

Notice that the method implementation calls the `super.addChild()` method. The call to `super.addChild()` causes Flex to invoke the superclass's `addChild()` method to perform the operation. Your new functionality to open the Alert box occurs after the `super.addChild()` method.

You might have to use `super()` to call the base class method before your code, after your code, or not at all. The location is determined by your requirements. To add functionality to the method, you call `super()` before your code. To replace the base class method, you do not call `super()` at all.

The following example uses this component in an application:

```
<?xml version="1.0"?>
<!-- as/MainHBoxWithAlert.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <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>

    <MyComp:HBoxWithAlert id="myHBox">
    </MyComp:HBoxWithAlert>

    <mx:Button label="Add Button" click="addButton();"/>

</mx:Application>
```

Initializing inherited properties with tag attributes in MXML

In an MXML component, you can initialize the value of any inherited public, writable property by defining a child tag of the MXML component with an `id` property that matches the name of the inherited property. For example, you define a custom Panel component based on the Flex Panel container, named MyPanel.as, as the following example shows:

```
package myComponents
{
    import mx.containers.Panel;
    import mx.controls.Text;
    import mx.controls.TextInput;

    public class MyPanel extends Panel {

        // Define public variables for two child components.
        public var myInput:TextInput;
        public var myOutput:TextInput;

        public function MyPanel() {
            super();
        }

        // Copy the text from one child component to another.
        public function xfer():void {
            myOutput.text = myInput.text;
        }
    }
}
```

In this example, the MyPanel component defines two variables corresponding to TextInput controls. You then create a custom MXML component, named MyPanelComponent.mxml, based on MyPanel.as, as the following example shows:

```
<?xml version="1.0"?>
<!-- myPanelComponent.mxml -->
<MyComps:MyPanel xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComps="myComponents.*">

      <mx:TextInput id="myInput"/>
      <mx:TextInput id="myOutput"/>

</MyComps:MyPanel>
```

Notice that the value of the `id` property for the two TextInput controls matches the variable names of the properties defined in the MyPanel component. Therefore, Flex initializes the inherited properties with the TextInput controls that you defined in MXML. This technique for initializing properties can be referred to as *code behind*.

You can use your custom component in the following Flex application:

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

   <MyComps:MyPanelComponent id="myP"/>

   <mx:Button label="Copy" click="myP.xfer();"/>

</mx:Application>
```

If the value of the `id` property of a TextInput control does not match an inherited property name, Flex creates a property of the component, where the `id` property defines the name of the new property.

To support initialization from MXML, an inherited property must have the following characteristics:

- The inherited property must be public.

  If you try to initialize a nonpublic inherited property, the Flex compiler issues an error.

- The inherited property must be writable.

  If you try to initialize a constant, or a property defined by a getter method without a corresponding setter method, the Flex compiler issues an error.

- The data type of the value that you specify to the inherited property must by compatible with the data type of the property.

  If you try to initialize a property with a value of an incompatible data type, the Flex compiler issues an error.

Defining events in ActionScript components

Flex components dispatch their own events and listen to other events. An object that wants to know about another object's events registers as a listener with that object. When an event occurs, the object dispatches the event to all registered listeners.

The core class of the Flex architecture, mx.core.UIComponent, defines core events, such as `updateComplete`, `resize`, `move`, `creationComplete`, and others that are fundamental to all components. Subclasses of these classes inherit and dispatch these events.

Custom components that extend existing Flex classes inherit all the events of the superclass. If you extend the Button class to create the MyButton class, you can use the events inherited from the Button class, such as `mouseOver` or `creationComplete`, as the following example shows:

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

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private function handleClick(eventObj:Event):void {
                // Define event listener.
            }
            private function handleCreationComplete(eventObj:Event):void {
                // Define event listener.
            }

        ]]>
    </mx:Script>

    <MyComp:MyButton
        click="handleClick(event);"
        creationComplete="handleCreationComplete(event);"/>

</mx:Application>
```

Your custom components can also define new events based on the requirements of your components. For example, the section "Using data binding with custom properties" on page 115 showed how to define a custom event so that properties of your component can work with the Flex data binding mechanism.

Handling predefined events within the custom component

The previous example showed a custom component, MyButton, dispatching two events. In that example, you defined the event listeners in the main application file.

Your custom component can also define event listeners within the component itself to handle the events internally. For example, "Defining public properties as variables" on page 110 defined event listeners for the `keyDown` and `creationComplete` events within the body of the component. This allows the component to handle those events internally.

Note: *Even though you define event listeners for the events in the component itself, your application can also register listeners for those events. The event listeners defined within the component execute before any listeners defined in the application.*

The example used the `creationComplete` event to access the default `fontSize` property of the component. You could not access this property in the constructor itself because Flex does not define it until after the component is created. For more information on the initialization order of a component, see "Advanced Visual Components in ActionScript" on page 129.

Dispatching custom events

Your ActionScript component can define custom events and use the predefined events. You use custom events to support data binding, to respond to user interactions, or to trigger actions by your component. For an example that uses events to support data binding, see "Using data binding with custom properties" on page 115.

For each custom event dispatched by your component, you must do the following:

1. Create an Event object describing the event.
2. (Optional) Use the `[Event]` metadata tag to make the event public so that other components can listen for it.
3. Dispatch the event by using the `dispatchEvent()` method.

To add information to the event object, you define a subclass of the flash.events.Event class to represent the event object. For more information on creating custom event classes, see "Custom Events" on page 23.

You might define some custom events that are used internally by your component, and are not intended to be recognized by the other components. For example, the following component defines a custom event, dispatches it, and handles it all within the component:

```
package myComponents
{
    import mx.controls.TextArea;
    import flash.events.Event;

    public class ModalText extends TextArea {

        public function ModalText() {
            super();

            // Register event listener.
            addEventListener("enableChanged", enableChangedListener);
        }

        public function enableInput(value:Boolean):void {
            // Method body.

            // Dispatch event.
            dispatchEvent(new Event("enableChanged"));
        }

        private function enableChangedListener(eventObj:Event):void {
            // Handle event.
        }
    }
}
```

In this example, the public method `enableInput()` lets the user enable or disable input to the control. When you call the `enableInput()` method, the component uses the `dispatchEvent()` method to dispatch the `enableChanged` event. The `dispatchEvent()` method has the following signature:

```
dispatchEvent(eventObj)
```

The *`eventObj`* argument is the event object that describes the event.

If you want an MXML component to be able to register a listener for the event, you must make the event known to the Flex compiler by using the `[Event]` metadata tag. For each public event that your custom component dispatches, you add an `[Event]` metadata keyword before the class definition that defines that event, as the following example shows:

```
[Event(name="enableChanged", type="flash.events.Event")]
public class ModalText extends TextArea {
    ...
}
```

If you do not identify an event in the class file with the `[Event]` metadata tag, the compiler generates an error when an MXML component attempts to register a listener for that event. Any component can register an event listener for the event in ActionScript using the `addEventListener()` method, even if you omit the `[Event]` metadata tag.

You can then handle the event in MXML, as the following example shows:

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

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            private function handleEnableChanged(event:Event):void {
                myTA.text="Got Event";
            }
        ]]>
    </mx:Script>

    <MyComps:ModalTextEvent id="myMT"
        enableChanged="handleEnableChanged(event);"/>

    <mx:Button click="myMT.enableInput(true);"/>
    <mx:TextArea id="myTA"/>
</mx:Application>
```

Applying styles to custom components

Style properties define the look of a component, from the size of the fonts used to the color of the background. Your custom ActionScript components inherit all of the styles of the base class, so you can set them in the same way as for that base class.

To change style properties in custom components, use the `setStyle()` method in the component's constructor. This applies the same style to all instances of the component, but users of the component can override the settings of the `setStyle()` method in MXML tags. Any style properties that are not set in the component's class file are inherited from the component's superclass.

The following ActionScript class file sets the `color` and `borderColor` styles of the BlueButton control:

```
package myComponents
{
    // as/myComponents/BlueButton.as
    import mx.controls.Button;

    public class BlueButton extends Button
    {

        public function BlueButton() {
            super();

            // Set the label text to blue.
            setStyle("color", 0x0000FF);

            // Set the borderColor to blue.
            setStyle("borderColor", 0x0000FF);
        }
    }
}
```

The following MXML file uses the BlueButton control with the default `color` and `borderColor` styles set in your component's class file:

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

    <MyComps:BlueButton label="Submit"/>

</mx:Application>
```

Setting the styles in a constructor does not prevent users of the component from changing the style. For example, the user could still set their own value for the `color` style, as the following example shows:

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

    <MyComps:BlueButton label="Submit" color="0xFF0000"/>

</mx:Application>
```

In addition to setting the `color` property, you can set the font face, font size, and other style properties. For more information on the available style properties, see the parent control's class information.

You can also define new style properties for your components. For more information, see "Advanced Visual Components in ActionScript" on page 129.

Applying styles from a defaults.css file

If you package your component in a SWC file, you can define a global style sheet, named defaults.css, in the SWC file. The defaults.css file defines the default style settings for all of the components defined in the SWC file.

For more information, see "Applying styles from a defaults.css file" on page 74.

Chapter 9: Advanced Visual Components in ActionScript

You can create advanced visual components for use in Adobe® Flex® applications.

Topics

About creating advanced components

Simple visual components are subclasses of existing Flex components that modify the appearance of the component by using skins or styles, or add new functionality to the component. For example, you add a new event type to a Button control, or modify the default styles or skins of a DataGrid control. For more information, see “Simple Visual Components in ActionScript” on page 105.

In advanced components, you typically perform the following actions:

- Modify the visual appearance or visual characteristics of an existing component.
- Create a composite component that encapsulates two or more components within it.
- Create a component by creating a subclass of the UIComponent class.

You usually create a component as a subclass of an existing class. For example, to create a component that is based on the Button control, you create a subclass of the mx.controls.Button class. To make your own component, you create a subclass of the mx.core.UIComponent class.

About overriding protected UIComponent methods

All Flex visual components are subclasses of the UIComponent class. Therefore, visual components inherit the methods, properties, events, styles, and effects defined by the UIComponent class.

To create an advanced visual component, you must implement a class constructor. Also, you optionally override one or more of the following protected methods of the UIComponent class:

UIComponent method	**Description**
`commitProperties()`	Commits any changes to component properties, either to make the changes occur at the same time or to ensure that properties are set in a specific order. For more information, see “Implementing the commitProperties() method” on page 138.
`createChildren()`	Creates any child components of the component. For example, the ComboBox control contains a TextInput control and a Button control as child components. For more information, see “Implementing the createChildren() method” on page 137.
`layoutChrome()`	Defines the border area around the container for subclasses of the Container class. For more information, see “Implementing the layoutChrome() method” on page 145.
`measure()`	Sets the default size and default minimum size of the component. For more information, see “Implementing the measure() method” on page 142.
`updateDisplayList()`	Sizes and positions the children of the component on the screen based on all previous property and style settings, and draws any skins or graphic elements used by the component. The parent container for the component determines the size of the component itself. For more information, see “Implementing the updateDisplayList() method” on page 146.

Component users do not call these methods directly; Flex calls them as part of the initialization process of creating a component, or when other method calls occur. For more information, see “About the component instantiation life cycle” on page 131.

About the invalidation methods

During the lifetime of a component, your application might modify the component by changing its size or position, modifying a property that controls its display, or modifying a style or skin property of the component. For example, you might change the font size of the text displayed in a component. As part of changing the font size, the component’s size might also change, which requires Flex to update the layout of the application. The layout operation might require Flex to invoke the `commitProperties()`, `measure()`, `layoutChrome()`, and the `updateDisplayList()` methods of your component.

Your application can programmatically change the font size of a component much faster than Flex can update the layout of an application. Therefore, you should only want to update the layout after you are sure that you've determined the final value of the font size.

In another scenario, when you set multiple properties of a component, such as the `label` and `icon` properties of a Button control, you want the `commitProperties()`, `measure()`, and `updateDisplayList()` methods to execute only once, after all properties are set. You do not want these methods to execute when you set the `label` property, and then execute again when you set the `icon` property.

Also, several components might change their font size at the same time. Rather than updating the application layout after each component changes its font size, you want Flex to coordinate the layout operation to eliminate any redundant processing.

Flex uses an invalidation mechanism to synchronize modifications to components. Flex implements the invalidation mechanism as a set of methods that you call to signal that something about the component has changed and requires Flex to call the component's `commitProperties()`, `measure()`, `layoutChrome()`, or `updateDisplayList()` methods.

The following table describes the invalidation methods:

Invalidation method	**Description**
`invalidateProperties()`	Marks a component so that its `commitProperties()` method gets called during the next screen update.
`invalidateSize()`	Marks a component so that its `measure()` method gets called during the next screen update.
`invalidateDisplayList()`	Marks a component so that its `layoutChrome()` and `updateDisplayList()` methods get called during the next screen update.

When a component calls an invalidation method, it signals to Flex that the component must be updated. When multiple components call invalidation methods, Flex coordinates updates so that they all occur together during the next screen update.

Typically, component users do not call the invalidation methods directly. Instead, they are called by the component's setter methods, or by any other methods of a component class as necessary. For more information and examples, see "Implementing the commitProperties() method" on page 138.

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 in ActionScript and adds it to a container:

```
// Create a Box container.
var boxContainer:Box = new Box();
// Configure the Box container.

// Create a Button control.
var b:Button = new Button()
// Configure the button control.
b.label = "Submit";
...
// Add the Button control to the Box container.
boxContainer.addChild(b);
```

The following steps show what occurs when you execute the code to create the Button control, and add the control to the Box container:

1 You call the component's constructor, as the following code shows:

   ```
   // Create a Button control.
   var b:Button = new Button()
   ```

2 You configure the component by setting its properties, as the following code shows:

   ```
   // Configure the button control.
   b.label = "Submit";
   ```

   Component setter methods might call the `invalidateProperties()`, `invalidateSize()`, or `invalidateDisplayList()` methods.

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.
   boxContainer.addChild(b);
   ```

   Flex performs the following actions:

   a Sets the `parent` property for the component to reference its parent container.

   b Computes the style settings for the component.

   c Dispatches the `preinitialize` event on the component.

   d Calls the component's `createChildren()` method.

   e Calls the `invalidateProperties()`, `invalidateSize()`, and `invalidateDisplayList()` methods to trigger later calls to the `commitProperties()`, `measure()`, or `updateDisplayList()` methods during the next `render` event.

      The only exception to this rule is that Flex does not call the `measure()` method when the user sets the height and width of the component.

   - **f** Dispatches the `initialize` event on the component. At this time, all of the component's children are initialized, but the component has not been sized or processed for layout. You can use this event to perform additional processing of the component before it is laid out.
   - **g** Dispatches the `childAdd` event on the parent container.
   - **h** Dispatches the `initialize` event on the parent container.

4 During the next `render` event, Flex performs the following actions:

   - **a** Calls the component's `commitProperties()` method.
   - **b** Calls the component's `measure()` method.
   - **c** Calls the component's `layoutChrome()` method.
   - **d** Calls the component's `updateDisplayList()` method.
   - **e** Dispatches the `updateComplete` event on the component.

5 Flex dispatches additional `render` events if the `commitProperties()`, `measure()`, or `updateDisplayList()` methods call the `invalidateProperties()`, `invalidateSize()`, or `invalidateDisplayList()` methods.

6 After the last `render` event occurs, Flex performs the following actions:

   - **a** Makes the component visible by setting the `visible` property to `true`.
   - **b** Dispatches the `creationComplete` event on the component. The component is sized and processed for layout. This event is only dispatched once when the component is created.
   - **c** Dispatches the `updateComplete` event on the component. Flex dispatches additional `updateComplete` events whenever the layout, position, size, or other visual characteristic of the component changes and the component is updated for display.

Most of the work for configuring a component occurs when you add the component to a container by using the `addChild()` method. That is because until you add the component to a container, Flex cannot determine its size, set inheriting style properties, or draw it on the screen.

You can also define your application in MXML, as the following example shows:

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

The sequence of steps that Flex executes when creating a component in MXML are equivalent to the steps described for ActionScript.

You can remove a component from a container by using the `removeChild()` method. If there are no references to the component, it is eventually deleted from memory by the garbage collection mechanism of Adobe® Flash® Player or Adobe® AIR™.

About the steps for creating a component

When you implement a component, you override component methods, define new properties, dispatch new events, or perform any other customizations required by your application.

To implement your component, follow these general steps:

1. If necessary, create any skins for the component.
2. Create an ActionScript class file.
   - **a** Extend one of the base classes, such as UIComponent or another component class.
   - **b** Specify properties that the user can set by using an MXML tag property.
   - **c** Embed any graphic and skin files.
   - **d** Implement the constructor.
   - **e** Implement the `UIComponent.createChildren()` method.
   - **f** Implement the `UIComponent.commitProperties()` method.
   - **g** Implement the `UIComponent.measure()` method.
   - **h** Implement the `UIComponent.layoutChrome()` method.
   - **i** Implement the `UIComponent.updateDisplayList()` method.
   - **j** Add properties, methods, styles, events, and metadata.
3. Deploy the component as an ActionScript file or as a SWC file.

For more information about MXML tag properties and embedding graphic and skin files, see “Simple Visual Components in ActionScript” on page 105.

You do not have to override all component methods to define a new component. You only override the methods required to implement the functionality of your component. If you create a subclass of an existing component, such as Button control or VBox container, you must implement the methods necessary for you to add any new functionality to the component.

For example, you can implement a custom Button control that uses a new mechanism for defining its default size. In that case, you only need to override the `measure()` method. For an example, see “Implementing the measure() method” on page 142.

Or you might implement a new subclass of the VBox container. Your new subclass uses all of the existing sizing logic of the VBox class, but changes the layout logic of the class to lay out the container children from the bottom of the container to the top, rather than from the top down. In this case, you only need to override the `updateDisplayList()` method. For an example, see "Implementing the updateDisplayList() method" on page 146.

About interfaces

Flex uses interfaces to divide the basic functionality of components into discrete elements so that they can be implemented piece by piece. For example, to make your component focusable, it must implement the IFocusable interface; to let it participate in the layout process, it must implement ILayoutClient interface.

To simplify the use of interfaces, the UIComponent class implements all of the interfaces defined in the following table, except for the IFocusManagerComponent and IToolTipManagerClient interfaces. However, many subclasses of UIComponent implement the IFocusManagerComponent and IToolTipManagerClient interfaces.

Therefore, if you create a subclass of the class or subclass of UIComponent, you do not have to implement these interfaces. But, if you create a component that is not a subclass of UIComponent, and you want to use that component in Flex, you might have to implement one or more of these interfaces.

***Note:** For Flex, Adobe recommends that all of your components extend the UIComponent class or a class that extends UIComponent.*

The following table lists the main interfaces implemented by Flex components:

Interface	**Use**
IChildList	Indicates the number of children in a container.
IDeferredInstantiationUIComponent	Indicates that a component or object can effect deferred instantiation.
IFlexDisplayObject	Specifies the interface for skin elements.
IFocusManagerComponent	Indicates that a component or object is focusable, which means that the components can receive focus from the FocusManager. The UIComponent class does not implement IFocusable because some components are not intended to receive focus.
IInvalidating	Indicates that a component or object can use the invalidation mechanism to perform delayed, rather than immediate, property commitment, measurement, and drawing or layout.
ILayoutManagerClient	Indicates that a component or object can participate in the LayoutManager's commit, measure, and update sequence.

Interface	Use
IPropertyChangeNotifier	Indicates that a component supports a specialized form of event propagation.
IRepeaterClient	Indicates that a component or object can be used with the Repeater class.
IStyleClient	Indicates that the component can inherit styles from another object, and supports the `setStyle()` and `getStyle()` methods.
IToolTipManagerClient	Indicates that a component has a `toolTip` property, and therefore is monitored by the ToolTipManager.
IUIComponent	Defines the basic set of APIs that you must implement in order to be a child of layout containers and lists.
IValidatorListener	Indicates that a component can listen for validation events, and therefore show a validation state, such as a red border and error tooltips.

Implementing the component

When you create a custom component in ActionScript, you have to override the methods of the UIComponent class. You implement the basic component structure, the constructor, and the `createChildren()`, `commitProperties()`, `measure()`, `layoutChrome()`, and `updateDisplayList()` methods.

Basic component structure

The following example shows the basic structure of a Flex component:

```
package myComponents
{
public class MyComponent extends UIComponent
{
....
}
}
```

You must define your ActionScript custom components within a package. The package reflects the directory location of your component within the directory structure of your application.

The class definition of your component must be prefixed by the `public` keyword. A file that contains a class definition can have one, and only one, public class definition, although it can have additional internal class definitions. Place any internal class definitions at the bottom of your source file below the closing curly brace of the package definition.

Implementing the constructor

Your ActionScript class should define a public constructor method for a class that is a subclass of the UIComponent class, or a subclass of any child of the UIComponent class. The constructor has the following characteristics:

- No return type
- Should be declared public
- No arguments
- Calls the `super()` method to invoke the superclass' constructor

Each class can contain only one constructor method; ActionScript does not support overloaded constructor methods. For more information, see "Defining the constructor" on page 16.

Use the constructor to set the initial values of class properties. For example, you can set default values for properties and styles, or initialize data structures, such as Arrays.

Do not create child display objects in the constructor; you should use it only for setting initial properties of the component. If your component creates child components, create them in the `createChildren()` method.

Implementing the createChildren() method

A component that creates other components or visual objects within it is called a *composite component*. For example, the Flex ComboBox control contains a TextInput control to define the text area of the ComboBox, and a Button control to define the ComboBox arrow. Components implement the `createChildren()` method to create child objects (such as other components) in the component.

You do not call the `createChildren()` method directly; Flex calls it when the call to the `addChild()` method occurs to add the component to its parent. Notice that the `createChildren()` method has no invalidation method, which means that you do not have to call it a second time after the component is added to its parent.

For example, you might define a new component that consists of a Button control and a TextArea control, where the Button control enables and disables user input to the TextArea control. The following example creates the TextArea and Button controls:

```
// Declare two variables for the component children.
private var text_mc:TextArea;
private var mode_mc:Button;

override protected function createChildren():void {

    // Call the createChildren() method of the superclass.
    super.createChildren();
    // Test for the existence of the children before creating them.
    // This is optional, but do this so a subclass can create a different
    // child.
    if (!text_mc) {
        text_mc = new TextArea();
        text_mc.explicitWidth = 80;
        text_mc.editable = false;
        text_mc.addEventListener("change", handleChangeEvent);
        // Add the child component to the custom component.
        addChild(text_mc);
    }

    // Test for the existence of the children before creating them.
    if (!mode_mc) {
        mode_mc = new Button();
        mode_mc.label = "Toggle Editing";
        mode_mc.addEventListener("click", handleClickEvent);
        // Add the child component to the custom component.
        addChild(mode_mc);
    }
}
```

Notice in this example that the `createChildren()` method calls the `addChild()` method to add the child component. You must call the `addChild()` method for each child object.

After you create a child component, you can use properties of the child component to define its characteristics. In this example, you create the Button and TextArea controls, initialize them, and register event listeners for them. You could also apply skins to the child components. For a complete example, see “Example: Creating a composite component” on page 151.

Implementing the commitProperties() method

You use the `commitProperties()` method to coordinate modifications to component properties. Most often, you use it with properties that affect how a component appears on the screen.

Flex schedules a call to the `commitProperties()` method when a call to the `invalidateProperties()` method occurs. The `commitProperties()` method executes during the next `render` event after a call to the `invalidateProperties()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateProperties()` method.

Calls to the `commitProperties()` method occur before calls to the `measure()` method. This lets you set property values that the `measure()` method might use.

The typical pattern for defining component properties is to define the properties by using getter and setter methods, as the following example shows:

```
// Define a private variable for the alignText property.
private var _alignText:String = "right";

// Define a flag to indicate when the _alignText property changes.
private var bAlignTextChanged:Boolean = false;

// Define getter and setter methods for the property.
public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;

    // Trigger the commitProperties(), measure(), and updateDisplayList()
    // methods as necessary.
    // In this case, you do not need to remeasure the component.
    invalidateProperties();
    invalidateDisplayList();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // Handle alignment change
    }
}
```

As you can see in this example, the setter method modifies the property, calls the `invalidateProperties()` and `invalidateDisplayList()` methods, and then returns. The setter itself does not perform any calculations based on the new property value. This design lets the setter method return quickly, and leaves any processing of the new value to the `commitProperties()` method.

Changing the alignment of text in a control does not necessarily change the control's size. However, if it does, include a call to the `invalidateSize()` method to trigger the `measure()` method.

The main advantages of using the `commitProperties()` method are the following:

- To coordinate the modifications of multiple properties so that the modifications occur synchronously.

  For example, you might define multiple properties that control the text displayed by the component, such as the alignment of the text within the component. A change to either the text or the alignment property requires Flex to update the appearance of the component. However, if you modify both the text and the alignment, you want Flex to perform any calculations for sizing or positioning the component once, when the screen updates.

  Therefore, you use the `commitProperties()` method to calculate any values based on the relationship of multiple component properties. By coordinating the property changes in the `commitProperties()` method, you can reduce unnecessary processing overhead.

- To coordinate multiple modifications to the same property.

  You do not necessarily want to perform a complex calculation every time a user updates a component property. For example, users modify the `icon` property of the Button control to change the image displayed in the button. Calculating the label position based on the presence or size of an icon can be a computationally expensive operation that you want to perform only when necessary.

  To avoid this behavior, you use the `commitProperties()` method to perform the calculations. Flex calls the `commitProperties()` method when it updates the display. That means you perform the calculations once when Flex updates the screen, regardless of the number of times the property changed between screen updates.

The following example shows how you can handle two related properties in the `commitProperties()` method:

```
// Define a private variable for the text property.
private var _text:String = "ModalText";
private var bTextChanged:Boolean = false;

// Define the getter method.
public function get text():String {
        return _text;
}

//Define the setter method to call invalidateProperties()
// when the property changes.
public function set text(t:String):void {
    _text = t;
    bTextChanged = true;
    invalidateProperties();
    // Changing the text causes the control to recalculate its default size.
    invalidateSize();
    invalidateDisplayList();
}
```

```
// Define a private variable for the alignText property.
private var _alignText:String = "right";
private var bAlignTextChanged:Boolean = false;

public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;
    invalidateProperties();
    invalidateDisplayList();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flags indicate a change to both properties.
    if (bTextChanged && bAlignTextChanged) {
        // Reset flags.
        bTextChanged = false;
        bAlignTextChanged = false;

        // Handle case where both properties changed.
    }

    // Check whether the flag indicates a change to the text property.
    if (bTextChanged) {
        // Reset flag.
        bTextChanged = false;

        // Handle text change.
    }


    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // Handle alignment change.
    }
}
```

Implementing the measure() method

The `measure()` method sets the default component size, in pixels, and optionally sets the component's default minimum size.

Flex schedules a call to the `measure()` method when a call to the `invalidateSize()` method occurs. The `measure()` method executes during the next `render` event after a call to the `invalidateSize()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateSize()` method.

When you set a specific height and width of a component, Flex does not call the `measure()` method, even if you explicitly call the `invalidateSize()` method. That is, Flex calls the `measure()` method only if the `explicitWidth` property or the `explicitHeight` property of the component is `NaN`.

In the following example, because you explicitly set the size of the Button control, Flex does not call the `Button.measure()` method:

```
<mx:Button height="10" width="10"/>
```

In a subclass of an existing component, you might implement the `measure()` method only if you are performing an action that requires modification to the default sizing rules defined in the superclass. Therefore, to set a new default size, or perform calculations at run time to determine component sizing rules, implement the `measure()` method.

You set the following properties in the `measure()` method to specify the default size:

Properties	Description
`measuredHeight` `measuredWidth`	Specifies the default height and width of the component, in pixels. These properties are set to 0 until the `measure()` method executes. Although you can leave them set to 0, it makes the component invisible by default.
`measuredMinHeight` `measuredMinWidth`	Specifies the default minimum height and minimum width of the component, in pixels. Flex cannot set the size of a component smaller than its specified minimum size.

The `measure()` method only sets the default size of the component. In the `updateDisplayList()` method, the parent container of the component passes to it its actual size, which may be different than the default size.

Component users can also override the default size settings in an application by using the component in the following ways:

- Setting the `explicitHeight` and `exlicitWidth` properties
- Setting the `width` and `height` properties
- Setting the `percentHeight` and `percentWidth` properties

For example, you can define a Button control with a default size of 100 pixels wide and 50 pixels tall, and a default minimum size of 50 pixels by 25 pixels, as the following example shows:

```
package myComponents
{
    // asAdvanced/myComponents/DeleteTextArea.as
    import mx.controls.Button;

    public class BlueButton extends Button {

        public function BlueButton() {
            super();
        }

        override protected function measure():void {
            super.measure();

            measuredWidth=100;
            measuredMinWidth=50;
            measuredHeight=50;
            measuredMinHeight=25;
        }
    }
}
```

The following application uses this button in an application:

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainBlueButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <mx:VBox>
        <MyComp:BlueButton/>
        <mx:Button/>
    </mx:VBox>
</mx:Application>
```

In the absence of any other sizing constraints on the button, the VBox container uses the default size and default minimum size of the button to calculate its size at run time. For information on the rules for sizing a component, see “Introducing Containers” on page 419 in *Adobe Flex 3 Developer Guide*.

You can override the default size settings in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- asAdvanced/MainBlueButtonResize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >
    <mx:VBox>
        <MyComp:BlueButton width="50%"/>
        <mx:Button/>
    </mx:VBox>
</mx:Application>
```

In this example, you specify that the width of the button is 50% of the width of the VBox container. When 50% of the width of the container is smaller than the minimum width of the button, the button uses its minimum width.

Calculating default sizes

The example in "Implementing the measure() method" on page 142 uses static values for the default size and default minimum size of a component. Some Flex components use static sizes. For example, the TextArea control has a default size of 100 pixels wide by 44 pixels high, regardless of the text it contains. If the text is larger than the TextArea control, the control displays scroll bars.

Often, you set the default size based on characteristics of the component or information passed to the component. For example, the Button control's `measure()` method examines its label text, margin settings, and font characteristics to determine the control's default size.

In the following example, you override the `measure()` method of the TextArea control so that it examines the text passed to the control, and calculates the default size of the TextArea control to display the entire text string in a single line:

```
package myComponents
{
    // asAdvanced/myComponents/MyTextArea.as
    import mx.controls.TextArea;
    import flash.text.TextLineMetrics;

    public class MyTextArea extends TextArea
    {

        public function MyTextArea() {
            super();
        }

        // The default size is the size of the text plus a 10 pixel margin.
        override protected function measure():void {
            super.measure();

            // Calculate the default size of the control based on the
            // contents of the TextArea.text property.
            var lineMetrics:TextLineMetrics = measureText(text);
            // Add a 10 pixel border area around the text.
            measuredWidth = measuredMinWidth = lineMetrics.width + 10;
            measuredHeight = measuredMinHeight = lineMetrics.height + 10;
        }
    }
}
```

For text strings that are longer than the display area of your application, you can add logic to increase the height of the TextArea control to display the text on multiple lines. The following application uses this component:

```
<?xml version="1.0"?>
<!-- asAdvanced/MainMyTextArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <MyComp:MyTextArea id="myTA" text="This is a long text strring that would normally
cause a TextArea control to display scroll bars. But, the custom MyTextArea control
calcualtes its default size based on the text size."/>

    <mx:TextArea id="flexTA" text="This is a long text strring that would normally cause
a TextArea control to display scroll bars. But, the custom MyTextArea control calcualtes
its default size based on the text size."/>

</mx:Application>
```

Implementing the layoutChrome() method

The Container class, and some subclasses of the Container class, use the `layoutChrome()` method to define the border area around the container.

Flex schedules a call to the `layoutChrome()` method when a call to the `invalidateDisplayList()` method occurs. The `layoutChrome()` method executes during the next `render` event after a call to the `invalidateDisplayList()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateDisplayList()` method.

Typically, you use the RectangularBorder class to define the border area of a container. For example, you can create the RectangularBorder object, and add it as a child of the component in your override of the `createChildren()` method.

When you create a subclass of the Container class, you can use the `createChildren()` method to create the content children of the container; the content children are the child components that appear within the container. You then use `updateDisplayList()` to position the content children.

You typically use the `layoutChrome()` method to define and position the border area of the container, and any additional elements that you want to appear in the border area. For example, the Panel container uses the `layoutChrome()` method to define the title area of the panel container, including the title text and close button.

The primary reason for dividing the handling of the content area of a container from its border area is to handle the situation when the `Container.autoLayout` property is set to `false`. When the `autoLayout` property is set to `true`, measurement and layout of the container and of its children are done whenever the position or size of a container child changes. The default value is `true`.

When the `autoLayout` property is set to `false`, measurement and layout are done only once, when children are added to or removed from the container. However, Flex executes the `layoutChrome()` method in both cases. Therefore, the container can still update its border area even when the `autoLayout` property is set to `false`.

Implementing the updateDisplayList() method

The `updateDisplayList()` method sizes and positions the children of your component based on all previous property and style settings, and draws any skins or graphic elements that the component uses. The parent container for the component determines the size of the component itself.

A component does not appear on the screen until its `updateDisplayList()` method gets called. Flex schedules a call to the `updateDisplayList()` method when a call to the `invalidateDisplayList()` method occurs. The `updateDisplayList()` method executes during the next `render` event after a call to the `invalidateDisplayList()` method. When you use the `addChild()` method to add a component to a container, Flex automatically calls the `invalidateDisplayList()` method.

The main uses of the `updateDisplayList()` method are the following:

- To set the size and position of the elements of the component for display.

  Many components are made up of one or more child components, or have properties that control the display of information in the component. For example, the Button control lets you specify an optional icon, and use the `labelPlacement` property to specify where the button text appears relative to the icon.

  The `Button.updateDisplayList()` method uses the settings of the `icon` and `labelPlacement` properties to control the display of the button.

  For containers that have child controls, the `updateDisplayList()` method controls how those child components are positioned. For example, the `updateDisplayList()` method on the HBox container positions its children from left to right in a single row; the `updateDisplayList()` method for a VBox container positions its children from top to bottom in a single column.

  To size components in the `updateDisplayList()` method, you use the `setActualSize()` method, not the sizing properties, such as `width` and `height`. To position a component, use the `move()` method, not the `x` and `y` properties.

- To draw any visual elements necessary for the component.

  Components support many types of visual elements such as skins, styles, and borders. Within the `updateDisplayList()` method, you can add these visual elements, use the Flash drawing APIs, and perform additional control over the visual display of your component.

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

```
protected function updateDisplayList(unscaledWidth:Number,
    unscaledHeight:Number):void
```

The properties have the following values:

unscaledWidth Specifies the width of the component, in pixels, in the component's coordinates, regardless of the value of the scaleX property of the component. This is the width of the component as determined by its parent container.

unscaledHeight Specifies the height of the component, in pixels, in the component's coordinates, regardless of the value of the `scaleY` property of the component. This is the height of the component as determined by its parent container.

Scaling occurs in Flash Player or AIR, after `updateDisplayList()` executes. For example, a component with an `unscaledHeight` value of 100, and with a `scaleY` property of 2.0, appears 200 pixels high in Flash Player or AIR.

Overriding the layout mechanism of the VBox container

The VBox container lays out its children from the top of the container to the bottom, in the order in which the children are added to the container. The following example overrides the `updateDisplayList()` method, which causes the VBox container to layout its children from the bottom of the container to the top:

```
package myComponents
{
    // asAdvanced/myComponents/BottomUpVBox.as
    import mx.containers.VBox;
    import mx.core.EdgeMetrics;
    import mx.core.UIComponent;

    public class BottomUpVBox extends VBox
    {

        public function BottomUpVBox() {
            super();
        }

        override protected function updateDisplayList(unscaledWidth:Number,
            unscaledHeight:Number):void {
```

```
            super.updateDisplayList(unscaledWidth, unscaledHeight);

            // Get information about the container border area.
            // The usable area of the container for its children is the
            // container size, minus any border areas.
            var vm:EdgeMetrics = viewMetricsAndPadding;

            // Get the setting for the vertical gap between children.
            var gap:Number = getStyle("verticalGap");

            // Determine the y coordinate of the bottom of the usable area
            // of the VBox.
            var yOfComp:Number = unscaledHeight-vm.bottom;

            // Temp variable for a container child.
            var obj:UIComponent;

            for (var i:int = 0; i < numChildren; i++)
            {
                // Get the first container child.
                obj = UIComponent(getChildAt(i));

                // Determine the y coordinate of the child.
                yOfComp = yOfComp - obj.height;

                // Set the x and y coordinate of the child.
                // Note that you do not change the x coordinate.
                obj.move(obj.x, yOfComp);

                // Save the y coordinate of the child,
                // plus the vertical gap between children.
                // This is used to calculate the coordinate
                // of the next child.
                yOfComp = yOfComp - gap;
            }
        }
    }
}
```

In this example, you use the `UIComponent.move()` method to set the position of each child in the container. You can also use the `UIComponent.x` and `UIComponent.y` properties to set these coordinates. The difference is that the `move()` method changes the location of the component and then dispatches a `move` event when you call the method immediately; setting the `x` and `y` properties changes the location of the component and dispatches the event on the next screen update.

The following application uses this component:

```
<?xml version="1.0"?>
<!-- asAdvanced/MainBottomVBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

  <MyComp:BottomUpVBox>
    <mx:Label text="Label 1"/>
    <mx:Button label="Button 1"/>

    <mx:Label text="Label 2"/>
    <mx:Button label="Button 2"/>

    <mx:Label text="Label 3"/>
    <mx:Button label="Button 3"/>

    <mx:Label text="Label 4"/>
    <mx:Button label="Button 4"/>

  </MyComp:BottomUpVBox>
</mx:Application>
```

Drawing graphics in your component

Every Flex component is a subclass of the Flash Sprite class, and therefore inherits the `Sprite.graphics` property. The `Sprite.graphics` property specifies a Graphics object that you can use to add vector drawings to your component.

For example, in the `updateDisplayList()` method, you can use methods of the Graphics class to draw borders, rules, and other graphical elements:

```
override protected function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number):void {

    super.updateDisplayList(unscaledWidth, unscaledHeight);

    // Draw a simple border around the child components.
    graphics.lineStyle(1, 0x000000, 1.0);
    graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);
}
```

Making components accessible

A growing requirement for web content is that it should be accessible to people who have disabilities. Visually impaired people can use the visual content in Flash applications by using screen reader software, which provides an audio description of the material on the screen.

When you create a component, you can include ActionScript that enables the component and a screen reader for audio communication. When developers use your component to build an application in Flash, they use the Accessibility panel to configure each component instance.

Flash includes the following accessibility features:

- Custom focus navigation
- Custom keyboard shortcuts
- Screen-based documents and the screen authoring environment
- An Accessibility class

To enable accessibility in your component, add the following line to your component's class file:

```
mx.accessibility.ComponentName.enableAccessibility();
```

For example, the following line enables accessibility for the MyButton component:

```
mx.accessibility.MyButton.enableAccessibility();
```

For additional information about accessibility, see "Creating Accessible Applications" on page 1139 in *Adobe Flex 3 Developer Guide*.

Adding version numbers

When releasing components, you can define a version number. This lets developers know whether they should upgrade, and helps with technical support issues. When you set a component's version number, use the static variable `version`, as the following example shows:

```
static var version:String = "1.0.0.42";
```

***Note:** Flex does not use or interpret the value of the* `version` *property.*

If you create many components as part of a component package, you can include the version number in an external file. That way, you update the version number in only one place. For example, the following code imports the contents of an external file that stores the version number in one place:

```
include "../myPackage/ComponentVersion.as"
```

The contents of the ComponentVersion.as file are identical to the previous variable declaration, as the following example shows:

```
static var version:String = "1.0.0.42";
```

Best practices when designing a component

Use the following practices when you design a component:

- Keep the file size as small as possible.
- Make your component as reusable as possible by generalizing functionality.
- Use the Border class rather than graphical elements to draw borders around objects.
- Use tag-based skinning.
- Assume an initial state. Because style properties are on the object, you can set initial settings for styles and properties so your initialization code does not have to set them when the object is constructed, unless the user overrides the default state.

Example: Creating a composite component

Composite components are components that contain multiple components. They might be graphical assets or a combination of graphical assets and component classes. For example, you can create a component that includes a button and a text field, or a component that includes a button, a text field, and a validator.

When you create composite components, you should instantiate the controls inside the component's class file. Assuming that some of these controls have graphical assets, you must plan the layout of the controls that you are including, and set properties such as default values in your class file. You must also ensure that you import all the necessary classes that the composite component uses.

Because the class extends one of the base classes, such as UIComponent, and not a controls class like Button, you must instantiate each of the controls as children of the custom component and arrange them on the screen.

Properties of the individual controls are not accessible from the MXML author's environment unless you design your class to allow this. For example, if you create a component that extends the UIComponent class and uses a Button and a TextArea component, you cannot set the Button control's label text in the MXML tag because you do not directly extend the Button class.

Creating the component

This example component, called ModalText and defined in the file ModalText.as, combines a Button control and a TextArea control. You use the Button control to enable or disable text input in the TextArea control.

Defining event listeners for composite components

Custom components implement the `createChildren()` method to create children of the component, as the following example shows:

```
override protected function createChildren():void {
    super.createChildren();

    // Create and initialize the TextArea control.
    if (!text_mc) {
        text_mc = new TextArea();
        ...
        text_mc.addEventListener("change", handleChangeEvent);
        addChild(text_mc);
    }

    // Create and initialize the Button control.
    if (!mode_mc) {
        mode_mc = new Button();
        ...
        mode_mc.addEventListener("click", handleClickEvent);
        addChild(mode_mc);
    }
}
```

The `createChildren()` method also contains a call to the `addEventListener()` method to register an event listener for the `change` event generated by the TextArea control, and for the `click` event for the Button control. These event listeners are defined within the ModalText class, as the following example shows:

```
// Handle events that are dispatched by the children.
private function handleChangeEvent(eventObj:Event):void {
        dispatchEvent(new Event("change"));
}

// Handle events that are dispatched by the children.
private function handleClickEvent(eventObj:Event):void {
        text_mc.editable = !text_mc.editable;
}
```

You can handle an event dispatched by a child of a composite component in the component. In this example, the event listener for the Button control's `click` event is defined in the class definition to toggle the `editable` property of the TextArea control.

However, if a child component dispatches an event, and you want that opportunity to handle the event outside of the component, you must add logic to your custom component to propagate the event. Notice that the event listener for the `change` event for the TextArea control propagates the event. This lets you handle the event in your application, as the following example shows:

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

    <mx:Script>
        <![CDATA[

            import flash.events.Event;

            function handleText(eventObj:Event)
            {
                ...
            }
        ]]>
    </mx:Script>

    <MyComp:ModalText change="handleText(event);"/>
</mx:Application>
```

Creating the ModalText component

The following code example implements the class definition for the ModalText component. The ModalText component is a composite component that contains a Button control and a TextArea control. This control has the following attributes:

- You cannot edit the TextArea control by default.
- You click the Button control to toggle editing of the TextArea control.
- You use the `textPlacement` property of the control to make the TextArea appear on the right side or the left side of the control.
- Editing the `textPlacement` property of the control dispatches the `placementChanged` event.
- You use the `text` property to programmatically write content to the TextArea control.
- Editing the `text` property of the control dispatches the `textChanged` event.
- Editing the text in the TextArea control dispatches the `change` event.

- You can use both the `textPlacement` property or the `text` property as the source for a data binding expression.
- You can optionally use skins for the up, down, and over states of the Button control.

The following is an example MXML file that uses the ModalText control and sets the `textPlacement` property to `left`:

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainModalText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <MyComp:ModalText textPlacement="left" height="40"/>

</mx:Application>
```

You can handle the `placementChanged` event to determine when the `ModalText.textPlacement` property is modified, as the following example shows:

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainModalTextEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <mx:Script>
      <![CDATA[
        import flash.events.Event;

        private function placementChangedListener(event:Event):void {
          myEvent.text="placementChanged event occurred - textPlacement = "
              + myMT.textPlacement as String;
        }
      ]]>
    </mx:Script>

    <MyComp:ModalText id="myMT"
        textPlacement="left"
        height="40"
        placementChanged="placementChangedListener(event);"/>
    <mx:TextArea id="myEvent" width="50%"/>

    <mx:Label text="Change Placement" />
    <mx:Button label="Set Text Placement Right"
        click="myMT.textPlacement='right';" />
    <mx:Button label="Set Text Placement Left"
        click="myMT.textPlacement='left';" />
```

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

The following example shows the ModalText.as file that defines this control:

```
package myComponents
{
    // Import all necessary classes.
    import mx.core.UIComponent;
    import mx.controls.Button;
    import mx.controls.TextArea;
    import flash.events.Event;
    import flash.text.TextLineMetrics;

    // ModalText dispatches a change event when the text of the child
    // TextArea control changes, a textChanged event when you set the text
    // property of ModalText, and a placementChanged event
    // when you change the textPlacement property of ModalText.
    [Event(name="change", type="flash.events.Event")]
    [Event(name="textChanged", type="flash.events.Event")]
    [Event(name="placementChanged", type="flash.events.Event")]

    /*** a) Extend UIComponent. ***/
    public class ModalText extends UIComponent {

        /*** b) Implement the class constructor. ***/
        public function ModalText() {
            super();
        }


        /*** c) Define variables for the two child components. ***/
        // Declare two variables for the component children.
        private var text_mc:TextArea;
        private var mode_mc:Button;


        /*** d) Embed new skins used by the Button component. ***/
        // You can create a SWF file that contains symbols  with the names
        // ModalUpSkin, ModalOverSkin, and ModalDownSkin.
        // If you do not have skins, comment out these lines.
        [Embed(source="Modal2.swf", symbol="blueCircle")]
        public var modeUpSkinName:Class;

        [Embed(source="Modal2.swf", symbol="blueCircle")]
        public var modeOverSkinName:Class;

        [Embed(source="Modal2.swf", symbol="greenSquare")]
```

```
public var modeDownSkinName:Class;


/*** e) Implement the createChildren() method. ***/
// Test for the existence of the children before creating them.
// This is optional, but we do this so a subclass can create a
// different child instead.
override protected function createChildren():void {
    super.createChildren();

    // Create and initialize the TextArea control.
    if (!text_mc)
    {
        text_mc = new TextArea();
        text_mc.explicitWidth = 80;
        text_mc.editable = false;
        text_mc.text= _text;
        text_mc.addEventListener("change", handleChangeEvent);
        addChild(text_mc);
    }

    // Create and initialize the Button control.
    if (!mode_mc)
    {   mode_mc = new Button();
        mode_mc.label = "Toggle Editing Mode";
        // If you do not have skins available,
        // comment out these lines.
        mode_mc.setStyle('overSkin', modeOverSkinName);
        mode_mc.setStyle('upSkin', modeUpSkinName);
        mode_mc.setStyle('downSkin', modeDownSkinName);
        mode_mc.addEventListener("click", handleClickEvent);
        addChild(mode_mc);
    }
}


/*** f) Implement the commitProperties() method. ***/
override protected function commitProperties():void {
    super.commitProperties();

    if (bTextChanged) {
        bTextChanged = false;
        text_mc.text = _text;
        invalidateDisplayList();
    }
}
```

```
/*** g) Implement the measure() method. ***/
// The default width is the size of the text plus the button.
// The height is dictated by the button.
override protected function measure():void {
    super.measure();

    // Since the Button control uses skins, get the
    // measured size of the Button control.
    var buttonWidth:Number = mode_mc.getExplicitOrMeasuredWidth();
    var buttonHeight:Number = mode_mc.getExplicitOrMeasuredHeight();

    // The default and minimum width are the measuredWidth
    // of the TextArea control plus the measuredWidth
    // of the Button control.
    measuredWidth = measuredMinWidth =
        text_mc.measuredWidth + buttonWidth;

    // The default and minimum height are the larger of the
    // height of the TextArea control or the measuredHeight of the
    // Button control, plus a 10 pixel border around the text.
    measuredHeight = measuredMinHeight =
        Math.max(mode_mc.measuredHeight,buttonHeight) + 10;
}


/*** h) Implement the updateDisplayList() method. ***/
// Size the Button control to the size of its label text
// plus a 10 pixel border area.
// Size the TextArea to the remaining area of the component.
// Place the children depending on the setting of
// the textPlacement property.
override protected function updateDisplayList(unscaledWidth:Number,
        unscaledHeight:Number):void {
    super.updateDisplayList(unscaledWidth, unscaledHeight);

    // Subtract 1 pixel for the left and right border,
    // and use a 3 pixel margin on left and right.
    var usableWidth:Number = unscaledWidth - 8;

    // Subtract 1 pixel for the top and bottom border,
    // and use a 3 pixel margin on top and bottom.
    var usableHeight:Number = unscaledHeight - 8;

    // Calculate the size of the Button control based on its text.
```

```
        var lineMetrics:TextLineMetrics = measureText(mode_mc.label);
        // Add a 10 pixel border area around the text.
        var buttonWidth:Number = lineMetrics.width + 10;
        var buttonHeight:Number = lineMetrics.height + 10;
        mode_mc.setActualSize(buttonWidth, buttonHeight);

        // Calculate the size of the text
        // Allow for a 5 pixel gap between the Button
        // and the TextArea controls.
        var textWidth:Number = usableWidth - buttonWidth - 5;
        var textHeight:Number = usableHeight;
        text_mc.setActualSize(textWidth, textHeight);

        // Position the controls based on the textPlacement property.
        if (textPlacement == "left") {
            text_mc.move(4, 4);
            mode_mc.move(4 + textWidth + 5, 4);
        }
        else {
            mode_mc.move(4, 4);
            text_mc.move(4 + buttonWidth + 5, 4);
        }

        // Draw a simple border around the child components.
        graphics.lineStyle(1, 0x000000, 1.0);
        graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);
    }


    /*** i) Add methods, properties, and metadata. ***/
    // The general pattern for properties is to specify a private
    // holder variable.
    private var _textPlacement:String = "left";

    // Create a getter/setter pair for the textPlacement property.
    public function set textPlacement(p:String):void {
        _textPlacement = p;
        invalidateDisplayList();
        dispatchEvent(new Event("placementChanged"));
    }

    // The textPlacement property supports data binding.
    [Bindable(event="placementChanged")]
    public function get textPlacement():String {
        return _textPlacement;
    }
```

```
        private var _text:String = "ModalText";
        private var bTextChanged:Boolean = false;

        // Create a getter/setter pair for the text property.
        public function set text(t:String):void {
            _text = t;
            bTextChanged = true;
            invalidateProperties();
            dispatchEvent(new Event("textChanged"));
        }

        [Bindable(event="textChanged")]
        public function get text():String {
                return text_mc.text;
        }

        // Handle events that are dispatched by the children.
        private function handleChangeEvent(eventObj:Event):void {
                dispatchEvent(new Event("change"));
        }

        // Handle events that are dispatched by the children.
        private function handleClickEvent(eventObj:Event):void {
                text_mc.editable = !text_mc.editable;
        }
    }
}
```

Troubleshooting

I get an error "don't know how to parse…" when I try to use the component from MXML.

This means that the compiler could not find the SWC file, or the contents of the SWC file did not list the component. Ensure that the SWC file is in a directory that Flex searches, and ensure that your `xmlns` property is pointing to the right place. Try moving the SWC file to the same directory as the MXML file and setting the namespace to "*" as the following example shows:

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

For more information, see "Using the Flex Compilers" on page 125 in *Building and Deploying Adobe Flex 3 Applications*.

I get an error “xxx is not a valid attribute...” when I try to use the component from MXML.

Ensure that the attribute is spelled correctly. Also ensure that it is not private.

I don't get any errors, but nothing appears.

Verify that the component was instantiated. One way to do this is to put a Button control and a TextArea control in the MXML application and set the `text` property to the ID for the component when the button is clicked. For example:

```
<!-- This verifies whether a component was instantiated. -->
<zz:mycomponent id="foo"/>
<mx:TextArea id="output"/>
<mx:Button label="Print Output" click="output.text = foo.id;"/>
```

The component is instantiated properly but does not appear (1).

In some cases, helper classes are not ready by the time your component requires them. Flex adds classes to the application in the order that they must be initialized (base classes, and then child classes). However, if you have a static method that gets called as part of the initialization of a class, and that static method has class dependencies, Flex does not know to place that dependent class before the other class, because it does not know when that method is going to be called.

One possible remedy is to add a static variable dependency to the class definition. Flex knows that all static variable dependencies must be ready before the class is initialized, so it orders the class loading correctly.

The following example adds a static variable to tell the linker that class A must be initialized before class B:

```
public class A {

    static function foo():Number {
        return 5;
    }
}

public class B {
    static function bar():Number {
        return mx.example.A.foo();
    }

    static var z = B.bar();
    // Dependency
    static var ADependency:mx.example.A = mx.example.A;
}
```

The component is instantiated properly but does not appear (2).

Verify that the `measuredWidth` and `measuredHeight` properties are nonzero. If they are zero or `NaN`, ensure that you implemented the `measure()` method correctly.

You can also verify that the `visible` property is set to `true`. If `visible is false`, ensure that your component called the `invalidateDisplayList()` method.

The component is instantiated properly but does not appear (3).

It is possible that there is another class or SWC file that overrides your custom class or the symbols used in your component. Ensure that there are no naming conflicts.

Chapter 10: Custom Style Properties

Styles are useful for defining the look and feel of your Adobe® Flex® applications, including letting users set component skins. You can use them to change the appearance of a single component, or apply them across all components, including custom components.

Topics

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. Also, you can override individual properties for each control at a local, document, or global level, giving you great flexibility in controlling the appearance of your applications.

For more information, see “Using Styles and Themes” on page 589 in *Adobe Flex 3 Developer Guide*.

About inheritance in Cascading Style Sheets

When you implement a style property in an ActionScript component, that property is automatically inherited by any subclasses of your class, just as methods and properties are inherited. This type of inheritance is called object-oriented inheritance.

Some style properties also support Cascading Style Sheet (CSS) inheritance. CSS inheritance means that if you set the value of a style property on a parent container, a child of that container inherits the value of the property when your application runs. For example, if you define the `fontFamily` style as Times for a Panel container, all children of that container use Times for `fontFamily`, unless they override that property.

In general, color and text styles support CSS inheritance, regardless of whether they are set by using CSS or style properties. All other styles do not support CSS inheritance, unless otherwise noted.

If you set a style on a parent that does not support CSS inheritance, such as `textDecoration`, only the parent container uses that value, and not its children. There is an exception to the rules of CSS inheritance. If you use the global type 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 style inheritance, see "Using Styles and Themes" on page 589 in *Adobe Flex 3 Developer Guide*.

About setting styles

Flex provide several ways of setting component styles: using MXML tag attributes, calling the `setStyle()` method, and CSS.

Setting styles using MXML tag attributes

Component users can use MXML tag attributes to set a style property on a component. For example, the following code creates a TextArea control, and then sets the `backgroundColor` style of the component to blue (0x0000FF):

```
<mx:TextArea id="myTA" backgroundColor="0x0000FF"/>
```

Setting styles using the setStyle() method

Component users can use the `setStyle()` method to set a style property on a component. For example, the following code creates a TextArea control, and then sets the `backgroundColor` style of the component to blue (0x0000FF):

```
var myTA:TextArea=new TextArea();
myTA.setStyle('backgroundColor', 0x0000FF);
```

Setting styles using CSS

Component users can use the `<mx:Styles>` tag to set CSS styles in an MXML application, as the following example shows:

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

    <mx:Style>
        TextArea {backgroundColor: "0x0000FF"}
    </mx:Style>

    <mx:TextArea/>

</mx:Application>
```

You can also import an external CSS file, as the following example shows:

```
<mx:Style source="myStyle.css"/>
```

Overriding the styleChanged() method

When a user sets a style on a component, Flex calls the component's `styleChanged()` method, passing to it the name of the style being set. When you create a custom component, you can override the `UIComponent.styleChanged()` method to check the style name passed to it, and handle the change accordingly, as the following example shows:

```
var bBackgroundColor:Boolean=false;

override public function styleChanged(styleProp:String):void {

    super.styleChanged(styleProp);

    // Check to see if style changed.
    if (styleProp=="backgroundColor")
    {
        bBackgroundColor=true;
        invalidateDisplayList();
        return;
    }
}
```

The `styleChanged()` method first calls superclass' `styleChanged()` method to let the superclass handle the style change.

After the superclass gets a call to handle the style change, your component can detect that the user set the `backgroundColor` style, and handle it. By handling the style change after the superclass makes the change, you can override the way the superclass handles the style.

Notice that the method calls the `invalidateDisplayList()` method, which causes Flex to execute the component's `updateDisplayList()` method at the next screen update. Although you can detect style changes in the `styleChanged()` method, you still use the `updateDisplayList()` method to draw the component on the screen. For more information, see "Defining a style property" on page 167.

Typically, you use a flag to indicate that a style changed. In the `updateDisplayList()` method, you check the flag and update the component based on the new style setting, as the following example shows:

```
override protected function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number):void {

    super.updateDisplayList(unscaledWidth, unscaledHeight);
    // Check to see if style changed.
    if (bBackgroundColor==true)
    {
        // Redraw the component using the new style.
        ...
    }
}
```

By using flags to signal style updates to the `updateDisplayList()` method, the `updateDisplayList()` method has to perform only the updates based on the style changes; it may not have to redraw or recalculate the appearance of the entire component. For example, if you are changing only the border color of a component, it is more efficient to redraw only the border, rather than redrawing the entire component every time someone changes a style.

Example: Creating style properties

When you create a component, you might want to create a style property so that component users can configure it by using styles. For example, you create a component, named StyledRectangle, that uses a gradient fill pattern to define its color, as the following example shows:

This gradient is defined by two colors that you set by using a new style property called `fillColors`. The `fillColors` style takes an array of two colors that component users can set. The StyledRectangle.as class defines default colors for the `fillColors` style, but you can also set them, as the following example shows:

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

    <!-- Set style by using a CSS type selector. -->
    <mx:Style>
        StyledRectangle {fillColors: #FF00FF, #00FFFF}
    </mx:Style>

    <!-- By default, use the style defined by the CSS type selector. -->
    <MyComp:StyledRectangle id="mySR1"/>

    <!-- By default, use the style defined by the CSS type selector. -->
    <MyComp:StyledRectangle id="mySR2"/>

    <!-- Change the default style by using the setStyle() method. -->
    <mx:Button label="Set gradient"
        click="mySR2.setStyle('fillColors', [0x000000, 0xFFFFFF]);"/>

    <!-- Set fillColors in MXML. -->
    <MyComp:StyledRectangle id="mySR3" fillColors="[0x00FF00, 0xFFFFFF]"/>

</mx:Application>
```

In this example, the CSS type selector for the StyledRectangle component sets the initial values of the `fillColors` property to `#FF00FF` and `#00FFFF`. For the second StyledRectangle components, you use the `click` event of a Button control to change the `fillColor` style by using the `setStyle()` method. The third component sets the style property by using an MXML tag attribute.

Defining a style property

You define a style property for a component in the class definition.

1 Insert the `[Style]` metadata tag that defines the style before the class definition.

You insert the `[Style]` metadata tag before the class definition to define the MXML tag attribute for a style property. If you omit the `[Style]` metadata tag, the MXML compiler issues a syntax error when you try to set the property as an MXML tag attribute.

The `[Style]` metadata tag has the following syntax:

```
[Style(name="style_name"[,property="value",...])]
```

For more information, see "Metadata Tags in Custom Components" on page 33.

2 Override the `styleChanged()` method to detect changes to the property.

3 Override `updateDisplayList()` method to incorporate the style into the component display.

4 Define a static initializer to set the default value of the style property.

For more information, see "Setting default style values" on page 170.

The following code example defines the StyledRectangle component and the `fillColors` style:. It also defines a second style property named `alphas` that you can use to set the alpha values of the fill:

```
package myComponents
{
    // skinstyle/myComponents/StyledRectangle.as
    import mx.core.UIComponent;
    import mx.styles.CSSStyleDeclaration;
    import mx.styles.StyleManager;
    import flash.display.GradientType;

    // Insert the [Style] metadata tag to define the name, type
    // and other information about the style property for the
    // MXML compiler.
    [Style(name="fillColors",type="Array",format="Color",inherit="no")]
    [Style(name="alphas",type="Array",format="Number",inherit="no")]

    public class StyledRectangle extends UIComponent
    {
        // Define a static variable.
        private static var classConstructed:Boolean = classConstruct();

        // Define a static method.
        private static function classConstruct():Boolean {
            if (!StyleManager.getStyleDeclaration("StyledRectangle"))
            {
                // If there is no CSS definition for StyledRectangle,
                // then create one and set the default value.
                var myRectStyles:CSSStyleDeclaration = new CSSStyleDeclaration();
                myRectStyles.defaultFactory = function():void
                {
                    this.fillColors = [0xFF0000, 0x0000FF];
                    this.alphas = [0.5, 0.5];
                }
                StyleManager.setStyleDeclaration("StyledRectangle", myRectStyles, true);

            }
            return true;
        }

        // Constructor
        public function StyledRectangle() {
            super();
        }

        // Define a default size of 100 x 100 pixels.
        override protected function measure():void {
            super.measure();

            measuredWidth = measuredMinWidth = 100;
            measuredHeight = measuredMinHeight = 100;
        }
```

```
// Define the flag to indicate that a style property changed.
private var bStypePropChanged:Boolean = true;

// Define the variable to hold the current gradient fill colors.
private var fillColorsData:Array;

// Define the variable to hold the current alpha values.
private var alphasData:Array;

// Define the variable for additional control on the fill.
// You can create a style property for this as well.
private var ratios:Array = [0x00, 0xFF];

// Override the styleChanged() method to detect changes in your new style.
override public function styleChanged(styleProp:String):void {

    super.styleChanged(styleProp);

    // Check to see if style changed.
    if (styleProp=="fillColors" || styleProp=="alphas")
    {
        bStypePropChanged=true;
        invalidateDisplayList();
        return;
    }
}


// Override updateDisplayList() to update the component
// based on the style setting.
override protected function updateDisplayList(unscaledWidth:Number,
        unscaledHeight:Number):void {
    super.updateDisplayList(unscaledWidth, unscaledHeight);

    // Check to see if style changed.
    if (bStypePropChanged==true)
    {
        // Redraw gradient fill only if style changed.
        fillColorsData=getStyle("fillColors");
        alphasData=getStyle("alphas");

        graphics.beginGradientFill(GradientType.LINEAR,
            fillColorsData, alphasData, ratios);
        graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);

        bStypePropChanged=false;
```

```
                }
            }
        }
    }
```

Setting default style values

One of the issues that you have to decide when you create a style property for your component is how to set its default value. Setting a default value for a style property is not as simple as calling the `setStyle()` method in the component's constructor; you must take into consideration how Flex processes styles, and the order of precedence of styles.

When Flex compiles your application, Flex first examines any style definitions in the `<mx:Style>` tag, before it creates any components. Therefore, if you call `setStyle()` from within the component's constructor, which occurs after processing the `<mx:Style>` tag, you set the style property on each instance of the component; this overrides any conflicting CSS declarations in the `<mx:Style>` tag.

The easiest way to set a default value for a style property is to define a static initializer in your component. A static initializer is executed once, the first time Flex creates an instance of a component. In "Defining a style property" on page 167, you defined a static initializer, by using the `classConstructed` variable and the `classConstruct()` method, as part of the StyledRectangle.as class.

The `classConstruct()` method is invoked the first time Flex creates a StyledRectangle component. This method determines whether a style definition for the StyledRectangle class already exists, defined by using the `<mx:Style>` tag. If no style is defined, the `classConstruct()` method creates one, and sets the default value for the style property.

Therefore, if you omit the `<mx:Style>` tag from your application, the style definition is created by the `classConstruct()` method, as the following example shows:

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

    <MyComp:StyledRectangle/>

</mx:Application>
```

If you include the `<mx:Style>` tag, the `<mx:Style>` tag creates the default style definition, as the following example shows:

```
<?xml version="1.0"?>
<!-- skinstyle\MainRectCSSStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">
    <mx:Style>
        StyledRectangle {fillColors: #FF00FF, #00FFFF}
    </mx:Style>

    <MyComp:StyledRectangle/>

</mx:Application>
```

Defining a style property for a skin

Flex lets you set component skins by using style properties. Your new component might also support skins and therefore support setting skins by using style properties.

The mechanism for creating style properties to support skinning is the same as for creating other style properties. Setting a style property for a skin triggers a call to the `styleChanged()` method. The `styleChanged()` method detects the change to the skin, and performs any updates to the appearance of the component in the `updateDisplayList()` method.

The one difference when defining a style property for a skin is how you specify the `[Style]` metadata tag. When the style property corresponds to a skin, you specify `Class` as the value to the `type` property of the metadata tag, as the following example shows:

```
[Style(name="downSkin", type="Class", inherit="no")]
```

For more information on creating skins, see “Creating Skins” on page 689 in *Adobe Flex 3 Developer Guide*.

Chapter 11: Template Components

One way to create reusable components is to define them as template components. A template component defines properties with a general data type that lets the component user specify an object of a concrete data type when using the component. By using a general data type to define component properties, you create highly reusable components that can work with many different types of objects.

Topics

About template components

A standard component defines a property with a concrete data type, such as Number or String. The component user must then pass a value that exactly matches the property's data type or else Adobe® Flex® issues a compiler error.

A *template component* is a component in which one or more of its properties is defined with a general data type. This property serves as a slot for values that can be of the exact data type of the property, or of a value of a subclass of the data type.For example, to accept any Flex visual component as a property value, you define the data type of the property as UIComponent. To accept only container components, you define the data type of the property as Container.

When you use the template component in an application, the component user sets the property value to be an object with a concrete data type. You can think of the property as a placeholder for information, where it is up to the component user, rather than the component developer, to define the actual data type of the property.

The following example shows an application that uses a template component called MyTemplateComponent:

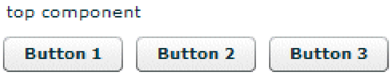
```
<?xml version="1.0"?>
<!-- templating/MainTemplateButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*"
    height="700" width="700">

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

        <MyComp:MyTemplateComponent id="myTComp1">
```

```
            <MyComp:topRow>
                <mx:Label text="top component"/>
            </MyComp:topRow>
            <MyComp:bottomRow>
                <mx:Button label="Button 1"/>
                <mx:Button label="Button 2"/>
                <mx:Button label="Button 3"/>
            </MyComp:bottomRow>
        </MyComp:MyTemplateComponent>
    </mx:Panel>
</mx:Application>
```

The MyTemplateComponent takes two properties:

- The `topRow` property specifies the single Flex component that appears in the top row of the VBox container.
- The `bottomRow` property specifies one or more Flex components that appear along the bottom row of the VBox container.

The implementation of the MyTemplateComponent consists of a VBox container that displays its children in two rows. The following image shows the output of this application:



The implementation of the `topRow` and `bottomRow` properties lets you specify any Flex component as a value, as the following example shows:

```
<?xml version="1.0"?>
<!-- templating/MainTemplateLink.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*"
    height="700" width="700">

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

        <MyComp:MyTemplateComponent id="myTComp2">
            <MyComp:topRow>
                <mx:TextArea text="top component"/>
            </MyComp:topRow>
            <MyComp:bottomRow>
                <mx:LinkButton label="Link 1"/>
                <mx:LinkButton label="Link 2"/>
                <mx:LinkButton label="Link 3"/>
            </MyComp:bottomRow>
        </MyComp:MyTemplateComponent>
    </mx:Panel>
</mx:Application>
```

In this example, the top component is a TextArea control, and the bottom components are two LinkButton controls.

Implementing a template component

The section "About template components" on page 173 shows an example of a template component named MyTemplateComponet. Flex provides you with two primary ways to create template components:

- Create properties with general data types, such as UIComponent or Container.
- Create properties with the type IDeferredInstance.

Using general data types in a template component

One way to implement the component MyTemplateComponent (see "About template components" on page 173) is to define the properties `topRow` and `bottomRow` as type UIComponent. Users of the component can specify any object to these properties that is an instance of the UIComponent class, or an instance of a subclass of UIComponent.

The following code shows the implementation of MyTemplateComponent:

```
<?xml version="1.0"?>
<!-- templating/myComponents/MyTemplateComponent.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="init();">

    <mx:Script>
        <![CDATA[

            import mx.containers.HBox;
            import mx.core.UIComponent;

            // Define a property for the top component.
            public var topRow:UIComponent;

            // Define an Array of properties for a row of components.
            // Restrict the type of the Array elements
            // to mx.core.UIComponent.
            [ArrayElementType("mx.core.UIComponent")]
            public var bottomRow:Array;
            private function init():void {
                // Add the top component to the VBox container.
                addChild(topRow);

                // Create an HBox container. This container
                // is the parent container of the bottom row of components.
                var controlHBox:HBox = new HBox();

                // Add the bottom row of components
                // to the HBox container.
                for (var i:int = 0; i < bottomRow.length; i++)
                    controlHBox.addChild(bottomRow[i]);

                // Add the HBox container to the VBox container.
                addChild(controlHBox);
            }
        ]]>
    </mx:Script>
</mx:VBox>
```

For the `bottomRow` property, you define it as an Array and include the `[ArrayElementType]` metadata tag to specify to the compiler that the data type of the Array elements is also UIComponent. For more information on the `[ArrayElementType]` metadata tag, see "Metadata Tags in Custom Components" on page 33.

Using IDeferredInstance in a template component

Deferred creation is a feature of Flex where Flex containers create only the controls that initially appear to the user. Flex then creates the container's other descendants if the user navigates to them. For more information, see "Improving Startup Performance" on page 91 in *Building and Deploying Adobe Flex 3 Applications*.

You can create a template component that also takes advantage of deferred creation. Rather than having Flex create your component and its properties when the application loads, you can define a component that creates its properties only when a user navigates to the area of the application that uses the component. This is especially useful for large components that may have many child components. Flex view states make use of this feature.

The following example shows an alternative implementation for the MyTemplateComponent component shown in the section "About template components" on page 173, named MyTemplateComponentDeferred.mxml, by defining the topRow and bottomRow properties to be of type IDeferredInstance:

```
<?xml version="1.0"?>
<!-- templating/myComponents/MyTemplateComponentDeferred.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="init();">
    <mx:Script>
        <![CDATA[

            import mx.containers.HBox;
            import mx.core.UIComponent;

            // Define a deferred property for the top component.
            public var topRow:IDeferredInstance;

            // Define an Array of deferred properties
            // for a row of components.
            [ArrayElementType("mx.core.IDeferredInstance")]
            public var bottomRow:Array;

            private function init():void {
                // Add the top component to the VBox container.
                // Cast the IDeferredInstance object to UIComponent
                // so that you can add it to the parent container.
                addChild(UIComponent(topRow.getInstance()));

                // Create an HBox container. This container
                // is the parent container of the bottom row of components.
                var controlHBox:HBox = new HBox();

                // Add the bottom row of components
                // to the HBox container.
                for (var i:int = 0; i < bottomRow.length; i++)
            controlHBox.addChild(UIComponent(bottomRow[i].getInstance()));

                // Add the HBox container to the VBox container.
                addChild(controlHBox);
            }
        ]]>
    </mx:Script>
</mx:VBox>
```

The IDeferredInstance interface defines a single method, `getInstance()`. Flex calls the `getInstance()` method to initialize a property when it creates an instance of the component. A subsequent call to the `getInstance()` method returns a reference to the property value.

In MXML, when the compiler encounters a value declaration for a property of type IDeferredInstance, instead of generating code to construct and assign the value to the property, the compiler generates code to construct and assign an IDeferredInstance implementation object, which then produces the value at run time.

You can pass any data type to a property of type IDeferredInstance. In the example in the section "About template components" on page 173, you pass a Label control to the `topRow` property, and three Button controls to the `bottomRow` property.

Notice in the example that the `addChild()` methods that take `topRow` and `bottomRow` as arguments cast them to UIComponent. This cast is necessary because the `addChild()` method can only add an object that implements the IUIComponent interface to a container, and the `DeferredInstance.getInstance()` method returns a value of type Object.

Defining properties using the IDeferredInstance interface

You can define component properties of type IDeferredInstance.

Defining a generic property

To define a generic property, one with no associated data type, you define its type as IDeferredInstance, as the following example shows:

```
// Define a deferred property for the top component.
public var topRow:IDeferredInstance;
```

The user of the component can then specify an object of any type to the property. It is your responsibility in the component implementation to verify that the value passed by the user is of the correct data type.

Restricting the data type of a property

You use the `[InstanceType]` metadata tag to specify the allowed data type of a property of type IDeferredInstance, as the following example shows:

```
// Define a deferred property for the top component.
[InstanceType("mx.controls.Label")]
public var topRow:IDeferredInstance;
```

The Flex compiler validates that users only assign values of the specified type to the property. In this example, if the component user sets the `topRow` property to a value of a type other than mx.controls.Label, the compiler issues an error message.

Defining an array of template properties

You can define an Array of template properties, as the following example shows:

```
// Define an Array of deferred properties for a row of components.
// Do not restrict the type of the component.
[ArrayElementType("mx.core.IDeferredInstance")]
public var bottomRow:Array;
```

```
// Define an Array of deferred properties for a row of components.
// Restrict the type of the component to mx.controls.Button.
[InstanceType("mx.controls.Button")]
[ArrayElementType("mx.core.IDeferredInstance")]
public var bottomRow:Array;
```

In the first example, you can assign a value of any data type to the `bottomRow` property. Each array element's `getInstance()` method is not called until the element is used.

In the second example, you can only assign values of type mx.controls.Button to it. Each Array element is created when the application loads. The following template component shows an alternative implementation of the MyTemplateComponent that restricts the type of components to be of type mx.controls.Button:

```
<?xml version="1.0"?>
<!-- templating/myComponents/MyTemplateComponentDeferredSpecific.mxml -->
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="init();">

    <mx:Script>
        <![CDATA[

            import mx.containers.HBox;
            import mx.core.UIComponent;

            [InstanceType("mx.controls.Label")]
            public var topRow:IDeferredInstance;

            // Define an Array of deferred properties
            // for a row of components.
            // Restrict the type of the component
            // to mx.controls.Button.
            [InstanceType("mx.controls.Button")]
            [ArrayElementType("mx.core.IDeferredInstance")]
            public var bottomRow:Array;

            private function init():void {
                addChild(UIComponent(topRow.getInstance()));

                var controlHBox:HBox = new HBox();
                for (var i:int = 0; i < bottomRow.length; i++)
            controlHBox.addChild(UIComponent(bottomRow[i].getInstance()));

                addChild(controlHBox);
            }
        ]]>
    </mx:Script>
</mx:VBox>
```

Part 4: Nonvisual Custom Components

Topics

Chapter 12: Custom Formatters

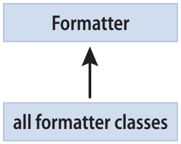
Adobe® Flex® includes several predefined formatters that you can use in your applications to format data. You also might have to extend the functionality of these predefined formatters, or create formatters for your specific application needs.

For more information on using formatters, see “Formatting Data” on page 1301 in *Adobe Flex 3 Developer Guide*.

Topics

Creating a custom formatter

You create a custom formatter by creating a class that extends the mx.formatters.Formatter base class, or by creating a class that extends one of the standard formatter classes, which all extend mx.formatters.Formatter. The following example shows the class hierarchy for formatters:



Like standard formatter classes, your custom formatter class must contain a public `format()` method that takes a single argument and returns a String that contains the formatted data. Most of the processing of your custom formatter occurs within the `format()` method.

Your custom formatter also might let the user specify which pattern formats the data. Where applicable, the Flex formatters, such as the ZipCodeFormatter, use a `formatString` property to pass a format pattern. Some Flex formatters, such as the NumberFormatter and CurrencyFormatter classes, do not have `formatString` properties, because they use a set of properties to configure formatting.

Creating a simple formatter

This example defines a simple formatter class that converts any String to all uppercase or all lowercase letters depending on the value passed to the `formatString` property. By default, the formatter converts a String to all uppercase.

```
package myFormatters
{
    // formatters/myFormatter/SimpleFormatter.as
    import mx.formatters.Formatter
    import mx.formatters.SwitchSymbolFormatter

    public class SimpleFormatter extends Formatter
    {
        // Declare the variable to hold the pattern string.
        public var myFormatString:String = "upper";

        // Constructor
        public function SimpleFormatter() {
            // Call base class constructor.
            super();
        }

        // Override format().
        override public function format(value:Object):String {
            // 1. Validate value - must be a nonzero length string.
            if( value.length == 0)
                {   error="0 Length String";
                    return ""
                }

            // 2. If the value is valid, format the string.
            switch (myFormatString) {
                case "upper" :
                    var upperString:String = value.toUpperCase();
                    return upperString;
                    break;
                case "lower" :
                    var lowerString:String = value.toLowerCase();
                    return lowerString;
                    break;
                default :
                    error="Invalid Format String";
                    return ""
            }
        }
    }
}
```

You can use this formatter in a Flex application, as the following example shows:

```
<?xml version="1.0" ?>
<!-- formatters/FormatterSimple.mxml -->

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

    <!-- Declare a formatter and specify formatting properties. -->
    <MyComp:SimpleFormatter id="upperFormat" myFormatString="upper" />

    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput id="myTI" />

    <mx:TextArea text="Your uppercase string is {upperFormat.format(myTI.text)}" />

</mx:Application>
```

The namespace declaration in the `<mx:Application>` tag specifies to use the `MyComp` prefix when referencing the formatter, and the location of the formatter's ActionScript file. That file is in the myFormatters subdirectory of the application, or in the default classpath of the application. For more information on deploying your formatters, see "Component Compilation" on page 51.

Handling errors in formatters

For all formatter classes, except for the SwitchSymbolFormatter class, when an error occurs, the formatter returns an empty string and writes a string that describes the error condition to the formatter's `error` property. The `error` property is inherited from the Formatter superclass.

In your application, you can test for an empty string in the result returned by the formatter. If detected, you can check the `error` property to determine the cause of the error. For an example that handles a formatter error, see "Formatting Data" on page 1301 in *Adobe Flex 3 Developer Guide*. For more information on the SwitchSymbolFormatter class, see "Using the SwitchSymbolFormatter class" on page 185.

Using the SwitchSymbolFormatter class

You can use the SwitchSymbolFormatter utility class when you create custom formatters. You use this class to replace placeholder characters in one string with numbers from a second string.

For example, you specify the following information to the SwitchSymbolFormatter class:

Format string The Social Security number is: ###-##-####"

Input string "123456789"

The SwitchSymbolFormatter class parses the format string and replaces each placeholder character with a number from the input string in the order in which the numbers are specified in the input string. The default placeholder character is the number sign (#). You can define a different placeholder character by passing it to the constructor when you create a SwitchSymbolFormatter object. For an example, see "Using a different placeholder character" on page 187.

The SwitchSymbolFormatter class creates the following output string from the Format and Input strings:

```
"The Social Security number is: 123-45-6789"
```

You pass the format string and input string to the `SwitchSymbolFormatter.formatValue()` method to create the output string, as the following example shows:

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

  <mx:Script>
    <![CDATA[

      import mx.formatters.SwitchSymbolFormatter;

      // Event handler to validate and format input.
      private function formatVal():void {

        var switcher:SwitchSymbolFormatter=new SwitchSymbolFormatter('#');

        formattedSCNumber.text =
        switcher.formatValue("Formatted Social Securty number: ###-##-#### ", scNum.text);
      }
    ]]>
  </mx:Script>

  <mx:Label text="Enter a 9 digit Social Security number with no separator characters:"/>
  <mx:TextInput id="scNum" text="" maxChars="9" width="50%"/>

  <mx:Button label="Format" click="formatVal();"/>
  <mx:TextInput id="formattedSCNumber" editable="false" width="75%"/>
</mx:Application>
```

You can mix alphanumeric characters and placeholder characters in this format string. The format string can contain any characters that are constant for all values of the numeric portion of the string. However, the input string for formatting must be numeric. The number of digits supplied in the source value must match the number of digits defined in the format string.

Using a different placeholder character

By default, the SwitchSymbolFormatter class uses a number sign (#) as the placeholder character to indicate a number substitution within its format string. However, sometimes you might want to include a number sign in your actual format string. Then, you must use a different symbol to indicate a number substitution slot within the format string. You can select any character for this alternative symbol as long as it doesn't appear in the format string.

For example, to use the ampersand character (&) as the placeholder, you create an instance of the SwitchSymbolFormatter class, as the following example shows:

```
var dataFormatter = new SwitchSymbolFormatter("&");
```

Handling errors with the SwitchSymbolFormatter class

Unlike other formatters, the SwitchSymbolFormatter class does not write its error messages into an `error` property. Instead, it is your responsibility to test for error conditions and return an error message if appropriate.

The custom formatter component in the following example formats nine-digit Social Security numbers by using the SwitchSymbolFormatter class:

```
package myFormatters
{
    // formatters/myFormatter/CustomSSFormatter.as
    import mx.formatters.Formatter
    import mx.formatters.SwitchSymbolFormatter

    public class CustomSSFormatter extends Formatter
    {
        // Declare the variable to hold the pattern string.
        public var formatString : String = "###-##-####";

        // Constructor
        public function CustomSSFormatter() {
            // Call base class constructor.
            super();
        }

        // Override format().
        override public function format( value:Object ):String {
            // Validate input string value - must be a 9-digit number.
            // You must explicitly check if the value is a number.
            // The formatter does not do that for you.
            if( !value  || value.toString().length != 9)
                {   error="Invalid String Length";
                    return ""
                }

            // Validate format string.
            // It must contain 9 number placeholders.
            var numCharCnt:int = 0;
            for( var i:int = 0; i<formatString.length; i++ )
                {
                    if( formatString.charAt(i) == "#" )
                    {   numCharCnt++;
                    }
                }

            if( numCharCnt != 9 )
            {
                error="Invalid Format String";
                return ""
            }

            // If the formatString and value are valid, format the number.
            var dataFormatter:SwitchSymbolFormatter =
                new SwitchSymbolFormatter();
            return dataFormatter.formatValue( formatString, value );
        }
    }
}
```

The following example uses this custom formatter in an application:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- formatters/FormatterSS.mxml -->

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

    <!-- Declare a formatter and specify formatting properties. -->
    <MyComp:CustomSSFormatter id="SSFormat"
        formatString="SS: #-#-#-#-#-#-#-#-#"/>

    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput text="Your SS number is {SSFormat.format('123456789')}"/>
```

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

Extending a Formatter class

You can extend the Formatter class to create a custom formatter, or any formatter class. The example in this section extends the ZipCodeFormatter class by allowing an extra format pattern: "#####*####".

In this example, if the user omits a format string, or specifies the default value of "#####*####", the formatter returns the ZIP code using the format "#####*####". If the user specifies any other format string, such as a five-digit string in the form "#####", the custom formatter calls the `format()` method in the superclass ZipCodeFormatter class to format the data.

```
package myFormatters
{
    // formatters/myFormatter/ExtendedZipCodeFormatter.as
    import mx.formatters.Formatter
    import mx.formatters.ZipCodeFormatter
    import mx.formatters.SwitchSymbolFormatter

    public class ExtendedZipCodeFormatter extends ZipCodeFormatter {

        // Constructor
        public function ExtendedZipCodeFormatter() {
            // Call base class constructor.
            super();
            // Initialize formatString.
            formatString = "#####*####";
        }

        // Override format().
        override public function format(value:Object):String {
            // 1. If the formatString is our new pattern,
            // then validate and format it.

            if( formatString == "#####*####" ){

                if( String( value ).length == 5 )
                    value = String( value ).concat("0000");

                if( String( value ).length == 9 ){
                    var dataFormatter:SwitchSymbolFormatter =
                        new SwitchSymbolFormatter();
                    return dataFormatter.formatValue( formatString, value );
                }
                else {
                    error="Invalid String Length";
                    return ""
                    }
                }

            // If the formatString is anything other than '#####*####,
            // call super and validate and format as usual using
            // the base ZipCodeFormatter.
            return super.format(value);
        }
    }
}
```

Notice that the ExtendedZipCodeFormatter class did not have to define a `formatString` property because it is already defined in its base class, ZipCodeFormatter.

The following example uses this custom formatter in an application:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- formatters/FormatterZC.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myFormatters.*">

    <!-- Declare a formatter and specify formatting properties. -->
    <MyComp:ExtendedZipCodeFormatter id="ZipCodeFormat"/>

    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput width="200"
        text="Your zipcode number is {ZipCodeFormat.format('123456789')}"/>

</mx:Application>
```

Chapter 13: Custom Validators

Data validators let you validate the data in an object. Adobe® Flex® supplies a number of standard validators that you can use in your application, but you can also define custom validators for your specific application needs.

For information on the standard validators, see “Validating Data” on page 1263 in *Adobe Flex 3 Developer Guide*.

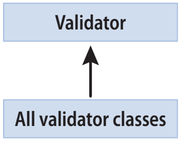
Topics

Validating data by using custom validators

The data that a user enters in a user interface might or might not be appropriate for 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 in a TextInput control.

Flex includes a set of validators for common types of user input data, such as ZIP codes, phone numbers, and credit cards. Although Flex supplies a number of commonly used validators, your application may require you to create custom validator classes. The mx.validators.Validator class is an ActionScript class that you can extend to add your own validation logic. Your classes can extend the functionality of an existing validator class, or you can implement new functionality in your custom validator class.

The following image shows the class hierarchy for validators:



About overriding the doValidation() method

Your custom validator class must contain an override of the protected `Validator.doValidation()` method that takes a single argument, `value`, of type Object, and returns an Array of ValidationResult objects. You return one ValidationResult object for each field that the validator examines and that fails the validation. For fields that pass the validation, you omit the ValidationResult object.

You do not have to create a ValidationResult object for fields that validate successfully. Flex creates those ValidationResult objects for you.

The base Validator class implements the logic to handle required fields by using the `required` property. When set to `true`, this property specifies that a missing or empty value in a user-interface control causes a validation error. To disable this verification, set this property to `false`.

In the `doValidation()` method of your validator class, you typically call the base class's `doValidation()` method to perform the verification for a required field. If the user did not enter a value, the base class issues a validation error stating that the field is required.

The remainder of the `doValidation()` method contains your custom validation logic.

About the ValidationResult class

The `doValidation()` method returns an Array of ValidationResult objects, one for each field that generates a validation error. The ValidationResult class defines several properties that let you record information about any validation failures, including the following:

errorCode A String that contains an error code. You can define your own error codes for your custom validators.

errorMessage A String that contains the error message. You can define your own error messages for your custom validators.

isError A Boolean value that indicates whether or not the result is an error. Set this property to `true`.

subField A String that specifies the name of the subfield associated with the ValidationResult object.

In your override of the `doValidation()` method, you can define an empty Array and populate it with ValidationResult objects as your validator encounters errors.

About the validate() method

You use the `Validator.validate()` method to programmatically invoke a validator from within a Flex application. However, you should never override this method in your custom validator classes. You need to override only the `doValidation()` method.

Example: Creating a simple validator

You can use the StringValidator class to validate that a string is longer than a minimum length and shorter than a maximum length, but you cannot use it to validate the contents of a string. This example creates a simple validator class that determines if a person is more than 18 years old based on their year of birth.

This validator extends the Validator base class, as the following example shows:

```
package myValidators
{
    import mx.validators.Validator;
    import mx.validators.ValidationResult;


    public class AgeValidator extends Validator {

        // Define Array for the return value of doValidation().
        private var results:Array;

        // Constructor.
        public function AgeValidator() {
            // Call base class constructor.
            super();
        }

        // Define the doValidation() method.
        override protected function doValidation(value:Object):Array {

            // Convert value to a Number.
            var inputValue:Number = Number(value);

            // Clear results Array.
            results = [];

            // Call base class doValidation().
            results = super.doValidation(value);
            // Return if there are errors.
            if (results.length > 0)
                return results;

            // Create a variable and initialize it to the current date.
            var currentYear:Date = new Date();

            // If input value is not a number, or contains no value,
            // issue a validation error.
            if (isNaN(inputValue) || !value )
```

```
                {
                    results.push(new ValidationResult(true, null, "NaN",
                        "You must enter a year."));
                    return results;
                }

                // If calculated age is less than 18, issue a validation error.
                if ((currentYear.getFullYear() - inputValue) < 18) {
                    results.push(new ValidationResult(true, null, "tooYoung",
                        "You must be 18."));
                    return results;
                }

                return results;
            }
        }
    }
```

This example first defines a public constructor that calls `super()` to invoke the constructor of its base class. The base class can perform the check to ensure that data was entered into a required field, if you set the `required` property of the validator to `true`.

Notice that the second argument of the constructor for the ValidationResult class is `null`. You use this argument to specify a subfield, if any, of the object being validated that caused the error. When you are validating a single field, you can omit this argument. For an example that validates multiple fields, see "Example: Validating multiple fields" on page 195.

You can use this validator in your Flex application, as the following example shows:

```
<?xml version="1.0" ?>
<!-- validators/MainAgeValidator.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myValidators.*">

    <MyComp:AgeValidator id="ageV"
        required="true"
        source="{birthYear}"
        property="text" />

    <mx:Form >
        <mx:FormItem label="Enter birth year: ">
            <mx:TextInput id="birthYear"/>
        </mx:FormItem>
        <mx:FormItem label="Enter birth year: ">
            <mx:Button label="Submit"/>
        </mx:FormItem>
    </mx:Form>
```

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

The `package` statement for your custom validator specifies that you should deploy it in a directory called `myValidators`. In the previous example, you place it in the subdirectory of the directory that contains your Flex application. Therefore, the namespace definition in your Flex application is `xmlns:MyComp="myValidators.*"`. For more information on deployment, see "Component Compilation" on page 51.

Example: Validating multiple fields

A validator can validate more than one field at a time. For example, you could create a custom validator called NameValidator to validate three input controls that represent a person's first, middle, and last names.

To create a validator that examines multiple fields, you can either define properties on the validator that let you specify the multiple input fields, as does the Flex DateValidator class, or you can require that the single item passed to the validator includes all of the fields to be validated.

In the following example, you use a NameValidator that validates an item that contains three fields named `first`, `middle`, and `last`:

```
<?xml version="1.0" ?>
<!-- validators/MainNameValidator.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myValidators.*">

    <mx:Model id="person">
        <name>
          <custName>
             <first>{firstInput.text}</first>
             <middle>{middleInput.text}</middle>
             <last>{lastInput.text}</last>
          </custName>
        </name>
    </mx:Model>

    <mx:TextInput id="firstInput"/>
    <mx:TextInput id="middleInput"/>
    <mx:TextInput id="lastInput"/>

    <MyComp:NameValidator id="nameVal"
        source="{person}" property="custName"
        listener="{firstInput}"/>
    <mx:Button label="Validate" click="nameVal.validate();"/>

</mx:Application>
```

This validator examines three input fields. You specify `firstInput` as the validation listener. Therefore, when a validation error occurs, Flex shows a validation error message on the first TextInput control.

You can implement the NameValidator class, as the following example shows:

```
package myValidators
{
    import mx.validators.Validator;
    import mx.validators.ValidationResult;

    public class NameValidator extends Validator {

        // Define Array for the return value of doValidation().
        private var results:Array;

        public function NameValidator () {
            super();
        }

        override protected function doValidation(value:Object):Array {

            var fName:String = value.first;
            var mName:String = value.middle;
            var lName:String = value.last;

            // Clear results Array.
            results = [];

            // Call base class doValidation().
            results = super.doValidation(value);
            // Return if there are errors.
            if (results.length > 0)
                return results;

            // Check first name field.
            if (fName == "" || fName == null) {
                results.push(new ValidationResult(true,
                    "first", "noFirstName", "No First Name."));
                return results;
            }

            // Check middle name field.
            if (mName == "" || mName == null) {
                results.push(new ValidationResult(true,
                    "middle", "noMiddleName", "No Middle Name."));
                return results;
            }

            // Check last name field.
            if (lName == "" || lName == null) {
                results.push(new ValidationResult(true,
                    "last", "noLastName", "No Last Name."));
                return results;
            }

            return results;
        }
    }
}
```

In this example, because you are using a single validator to validate three subfields of the Object passed to the validator, you include the optional second argument to the constructor for the ValidationResult class to specify the subfield that caused the validation error. This inclusion permits Flex to identify the input component that caused the error, and to highlight that component in the application.

The `doValidation()` method returns a validation error as soon as it detects the first validation error. You can modify `doValidation()` so that it examines all of the input fields before returning an error message, as the following example shows. This custom validator is named NameValidatorAllFields.as:

```
package myValidators
{
    import mx.validators.Validator;
    import mx.validators.ValidationResult;

    public class NameValidatorAllFields extends Validator {

        // Define Array for the return value of doValidation().
        private var results:Array;

        public function NameValidatorAllFields() {
            super();
        }

        override protected function doValidation(value:Object):Array {

            var fName:String = value.first;
            var mName:String = value.middle;
            var lName:String = value.last;
            // Clear results Array.
            results = [];

            // Call base class doValidation().
            results = super.doValidation(value);
            // Return if there are errors.
            if (results.length > 0)
                return results;

            // Check first name field.
            if (fName == "" || fName == null) {
                results.push(new ValidationResult(true,
                    "first", "noFirstName", "No First Name."));
            }

            // Check middle name field.
            if (mName == "" || mName == null) {
                results.push(new ValidationResult(true,
                    "middle", "noMiddleName", "No Middle Name."));
            }

            // Check last name field.
            if (lName == "" || lName == null) {
                results.push(new ValidationResult(true,
                    "last", "noLastName", "No Last Name."));
            }

            return results;
        }
    }
}
```

Notice that you remove the `return` statement from the body of the `if` statements so that the method contains only a single `return` statement. This modification allows you to detect three different validation errors at once.

Chapter 14: Effects

Behaviors let you add animation and motion to your application when some user or programmatic action occurs, where 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. 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.

Adobe® Flex® supplies a number of standard effects that you can use in your application. However, you also can define custom effects for your specific application needs.

For information on the standard effects, see "Using Behaviors" on page 545 in *Adobe Flex 3 Developer Guide*.

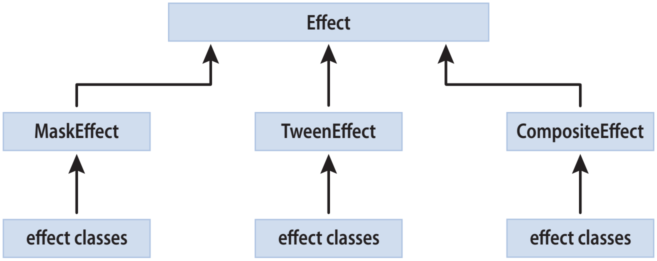
Topics

About creating a custom effect

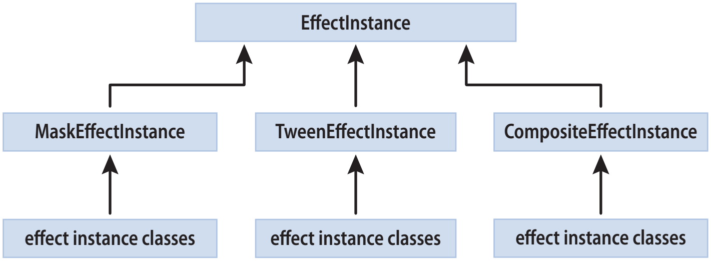
Flex implements effects by using an architecture in which each effect is represented by two classes: a factory class and an instance class. Therefore, to implement a custom effect, you create two classes: the factory class and the instance class.

You create a factory class by creating a subclass of the mx.effects.Effect class, or by creating a subclass of one of the subclasses of the mx.effects.Effect class. You create an instance class by creating a subclass of the mx.effects.EffectInstance class, or a subclass of one of the subclasses of the mx.effects.EffectInstance class.

The following image shows the class hierarchy for effects:



Factory classes



Instance classes

Defining factory and instance classes

To define a custom effect, you create two classes: the factory class and the 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.

About the effect base classes

You define effects by creating a subclass from the effects class hierarchy. Typically, you create a subclass from one of the following classes:

- mx.effects.Effect Create a subclass from this class for simple effects that do not require an effect to play over a period of time. For example, the Pause effect inserts a delay between two consecutive effects. You can also define a simple sound effect that plays an MP3 file.
- mx.effects.TweenEffect Create a subclass from this class to define an effect that plays over a period of time, such as an animation. For example, the Resize effect is a subclass of the TweenEffect class that modifies the size of its target over a specified duration.

About implementing your effects classes

You must override several methods and properties in your custom effect classes, and define any new properties and methods that are required to implement the effect. You can optionally override additional properties and methods based on the type of effect that you create.

The following table lists the methods and properties that you define in a factory class:

Factory method/property	Description
`constructor`	(Required) The class constructor. You typically call the `super()` method to invoke the superclass constructor to initialize the inherited items from the superclasses. Your constructor must take at least one optional argument, of type Object. This argument specifies the target component of the effect.
`Effect.initInstance()`	(Required) Copies properties of the factory class to the instance class. Flex calls this protected method from the `Effect.createInstance()` method; you do not have to call it yourself. In your override, you must call the `super.initInstance()` method.
`Effect.getAffectedProperties()`	(Required) Returns an Array of Strings, where each String is the name of a property of the target object that is changed by this effect. If the effect does not modify any properties, it should return an empty Array.
`Effect.instanceClass`	(Required) Contains an object of type Class that specifies the name of the instance class for this effect class. All subclasses of the Effect class must set this property, typically in the constructor.
`Effect.effectEndHandler()`	(Optional) Called when an effect instance finishes playing. If you override this method, ensure that you call the `super()` method.
`Effect.effectStartHandler()`	(Optional) Called when the effect instance starts playing. If you override this method, ensure that you call the `super()` method.
Additional methods and properties	(Optional) Define any additional methods and properties that the user requires to configure the effect.

The following table lists the methods and properties that you define in an instance class:

Instance method/property	Description
constructor	(Required) The class constructor. You typically call the `super()` method to invoke the superclass constructor to initialize the inherited items from the superclasses.
`EffectInstance.play()`	(Required) Invokes the effect. You must call `super.play()` from your override.
`EffectInstance.end()`	(Optional) Interrupts an effect that is currently playing, and jumps immediately to the end of the effect.
`EffectInstance.initEffect()`	(Optional) Called if the effect was triggered by the EffectManager. You rarely have to implement this method. For more information, see "Overriding the initEffect() method" on page 226.

Instance method/property	Description
`TweenEffectInstance.onTweenUpdate()`	(Required) Use when you create a subclass from TweenEffectInstance. A callback method called at regular intervals to implement a tween effect. For more information, see "Example: Creating a tween effect" on page 208.
`TweenEffectInstance.onTweenEnd()`	(Optional) Use when you create a subclass from TweenEffectInstance. A callback method called when the tween effect ends. You must call `super.onTweenEnd()` from your override. For more information, see "Example: Creating a tween effect" on page 208.
Additional methods and properties	(Optional) Define any additional methods and properties. These typically correspond to the public properties and methods from the factory class, and any additional properties and methods that you require to implement the effect.

Example: Defining a simple effect

To define a simple custom effect, you create a factory class from the Effect base class, and the instance class from the mx.effects.EffectInstance class. The following example shows an effect class that uses a Sound object to play an embedded MP3 file when a user action occurs. This example is a simplified version of the SoundEffect class that ships with Flex.

```
package myEffects
{
   // myEffects/MySound.as
   import mx.effects.Effect;
   import mx.effects.EffectInstance;
   import mx.effects.IEffectInstance;

   public class MySound extends Effect
   {
       // Define constructor with optional argument.
       public function MySound(targetObj:Object = null) {
           // Call base class constructor.
           super(targetObj);

           // Set instanceClass to the name of the effect instance class.
           instanceClass= MySoundInstance;
       }

       // This effect modifies no properties, so your
       // override of getAffectedProperties() method
       // returns an empty array.
       override public function getAffectedProperties():Array {
           return [];
       }
```

```
        // Override initInstance() method.
        override protected function initInstance(inst:IEffectInstance):void {
            super.initInstance(inst);
        }
    }
}
```

The `package` statement in your class specifies that you should deploy it in a directory called myEffects. In this example, you place it in the subdirectory of the directory that contains your Flex application. Therefore, the namespace definition in your Flex application is `xmlns:MyComp="myEffects.*"`. For more information on deployment, see "Component Compilation" on page 51.

To define your instance class, you create a subclass from the mx.effects.EffectInstance class. In the class definition, you must define a constructor and `play()` methods, and you can optionally define an `end()` method to stop the effect.

```
package myEffects
{
    // myEffects/MySoundInstance.as
    import mx.effects.EffectInstance;
    import flash.media.SoundChannel;
    import flash.media.Sound;

    public class MySoundInstance extends EffectInstance
    {

        // Embed the MP3 file.
        [Embed(source="sample.mp3")]
        [Bindable]
        private var sndCls:Class;

        // Define local variables.
        private var snd:Sound = new sndCls() as Sound;
        private var sndChannel:SoundChannel;

        // Define constructor.
        public function MySoundInstance(targetObj:Object) {
            super(targetObj);
        }

        // Override play() method.
        // Notice that the MP3 file is embedded in the class.
        override public function play():void {
            super.play();
            sndChannel=snd.play();
        }

        // Override end() method class to stop the MP3.
        override public function end():void {
            sndChannel.stop();
            super.end();
        }
    }
}
```

To use your custom effect class in an MXML file, you insert a tag with the same name as the factory class in the MXML file. You reference the custom effect the same way that you reference a standard effect.

The following example shows an application that uses the MySound effect:

```
<?xml version="1.0"?>
<!-- effects/MainSoundEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myEffects.*">

    <MyComp:MySound id="mySoundEffect" />

    <!-- Use the SoundEffect effect with a mouseOver trigger. -->
    <mx:Label text="play MP3" rollOverEffect="{mySoundEffect}" />

</mx:Application>
```

Example: Passing parameters to effects

To make your effects more robust, you often design them to let the user pass parameters to them. The example in this section modifies the sound effect from the previous section to take a parameter that specifies the MP3 file to play:

```
package myEffects
{
    // MySoundParam.as
    import mx.effects.Effect;
    import mx.effects.EffectInstance;
    import mx.effects.IEffectInstance;

    public class MySoundParam extends Effect
    {

       // Define a variable for the MP3 URL
       // and give it a default value.
       public var soundMP3:String=
           "http://localhost:8100/flex/assets/default.mp3";
```

```
        // Define constructor with optional argument.
        public function MySoundParam(targetObj:Object = null) {
            // Call base class constructor.
            super(targetObj);

            // Set instanceClass to the name of the effect instance class.
           instanceClass= MySoundParamInstance;
        }

        // Override getAffectedProperties() method to return an empty array.
        override public function getAffectedProperties():Array {
            return [];
        }

        // Override initInstance() method.
        override protected function initInstance(inst:IEffectInstance):void {
            super.initInstance(inst);
            // initialize the corresponding parameter in the instance class.
            MySoundParamInstance(inst).soundMP3 = soundMP3;
        }
    }
}
```

In the MySoundParam class, you define a variable named `soundMP3` that enables the user of the effect to specify the URL of the MP3 file to play. You also modify your override of the `initInstance()` method to pass the value of the `soundMP3` variable to the instance class.

Notice that the `getAffectedProperties()` method still returns an empty Array. That is because `getAffectedProperties()` returns the list of properties of the effect target that are modified by the effect, not the properties of the effect itself.

In your instance class, you define a property named soundMP3, corresponding to the property with the same name in the factory class. Then, you use it to create the URLRequest object to play the sound, as the following example shows:

```
package myEffects
{
    // MySoundParamInstance.as
    import mx.effects.EffectInstance;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;

    public class MySoundParamInstance extends EffectInstance
    {
        // Define local variables.
        private var s:Sound;
        private var sndChannel:SoundChannel;
        private var u:URLRequest;

        // Define a variable for the MP3 URL.
        public var soundMP3:String;

        // Define constructor.
        public function MySoundParamInstance(targetObj:Object) {
            super(targetObj);
        }

        // Override play() method.
        override public function play():void     {
            // You must call super.play() from within your override.
            super.play();
            s = new Sound();
            // Use the new parameter to specify the URL.
            u = new URLRequest(soundMP3);
            s.load(u);
            sndChannel=s.play();
        }

        // Override end() method to stop the MP3.
        override public function end():void  {
            sndChannel.stop();
            super.end();
        }
    }
}
```

You can now pass the URL of an MP3 to the effect, as the following example shows:

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

    <MyComps:MySoundParam id="mySoundEffect"
        soundMP3="http://localhost:8100/flex/assets/sample.mp3"/>

    <!-- Use the SoundEffect effect with a mouseOver trigger. -->
    <mx:Label text="play MP3" rollOverEffect="mySoundEffect"/>

</mx:Application>
```

About tween effects

Most Flex effects are implemented by using the tweening mechanism, where a *tween* defines a transition performed on a target object over a period of time. That transition could be a change in size, such as the Zoom or Resize effects perform; a change in visibility, such as the Fade or Dissolve effects perform; or other types of transitions.

You use the following classes to implement a tween effect:

- mx.effects.Tween A class used to implement tween effects. A Tween object accepts a start value, an end value, and an optional easing function. When you define tween effect classes, you create an instance of the Tween class in your override of the `Effect.play()` method.

  The Tween object invokes the `mx.effects.TweenEffect.onTweenUpdate()` callback method on a regular interval, passing the callback method an interpolated value between the start and end values. Typically, the callback method updates some property of the target component, causing that component's property to animate over time. For example, the Move effect modifies the `x` and `y` properties of the target component for the duration of the effect to show an animated movement.

  When the effect ends, the Tween object invokes the `mx.effects.TweenEffect.onTweenEnd()` callback method. This method performs any final processing before the effect terminates. You must call `super.onTweenEnd()` from your override.
- mx.effects.TweenEffect The base factory class for all tween effects. This class encapsulates methods and properties that are common among all Tween-based effects.
- mx.effects.effectClasses.TweenEffectInstance The instance class for all tween effects.

  When you define effects based on the TweenEffect class, you must override the `TweenEffectInstance.onTweenUpdate`() method, and optionally override the `TweenEffectInstance.onTweenEnd`() method.

The following example creates a tween effect. However, Flex supplies the AnimateProperty class that you can use to create a tween effect for a single property of the target component. For more information, see "Using Behaviors" on page 545 in *Adobe Flex 3 Developer Guide*.

Example: Creating a tween effect

In this example, you create a tween effect that rotates a component in a circle. This example implements a simplified version of the Rotate effect. The rotation is controlled by two parameters that are passed to the effect: `angleFrom` and `angleTo`:

```
package myEffects
{
   // myEffects/Rotation.as
   import mx.effects.TweenEffect;
   import mx.effects.EffectInstance;
   import mx.effects.IEffectInstance;

   public class Rotation extends TweenEffect
   {
       // Define parameters for the effect.
       public var angleFrom:Number = 0;
       public var angleTo:Number = 360;

       // Define constructor with optional argument.
       public function Rotation(targetObj:* = null) {
           super(targetObj);
           instanceClass= RotationInstance;
       }

       // Override getAffectedProperties() method to return "rotation".
       override public function getAffectedProperties():Array {
           return ["rotation"];
       }

       // Override initInstance() method.
       override protected function initInstance(inst:IEffectInstance):void {
           super.initInstance(inst);
           RotationInstance(inst).angleFrom = angleFrom;
           RotationInstance(inst).angleTo = angleTo;
       }
   }
}
```

In this example, the effect works by modifying the `rotation` property of the target component. Therefore, your override of the `getAffectedProperties()` method returns an array that contains a single element.

You derive your instance class from the TweenEffectInstance class, and override the `play()`, `onTweenUpdate()`, and `onTweenEnd()` methods, as the following example shows:

```
package myEffects
{
    // myEffects/RotationInstance.as
    import mx.effects.effectClasses.TweenEffectInstance;
    import mx.effects.Tween;

    public class RotationInstance extends TweenEffectInstance
    {
        // Define parameters for the effect.
        public var angleFrom:Number;
        public var angleTo:Number;

        public function RotationInstance(targetObj:*) {
            super(targetObj);
        }

        // Override play() method class.
        override public function play():void {
            // All classes must call super.play().
            super.play();
            // Create a Tween object. The tween begins playing immediately.
            var tween:Tween =
                createTween(this, angleFrom, angleTo, duration);
        }

        // Override onTweenUpdate() method.
        override public function onTweenUpdate(val:Object):void {
            target.rotation = val;
        }

        // Override onTweenEnd() method.
        override public function onTweenEnd(val:Object):void {
            // All classes that implement onTweenEnd()
            // must call    super.onTweenEnd().
            super.onTweenEnd(val);
        }
    }
}
```

In this example, the Tween object invokes the `onTweenUpdate()` callback method on a regular interval, passing it values between `angleFrom` and `angleTo`. At the end of the effect, the Tween object calls the `onTweenUpdate()` callback method with a value of `angleTo`. By invoking the `onTweenUpdate()` callback method at regular intervals throughout the duration of the effect, the target component displays a smooth animation as it rotates.

You use your new effect in an MXML application, as the following example shows:

```
<?xml version="1.0"?>
<!-- effects/MainRotation.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myEffects.*">

    <MyComp:Rotation id="Rotate90"
        angleFrom="0" angleTo="360"
        duration="1000"/>

    <mx:Image source="@Embed(source='../assets/myRotationImage.jpg')"
        mouseDownEffect="{Rotate90}"/>
```

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

In this example, you use the effect to rotate an image when the user clicks it.

Writing an effect for a transition

Transitions define how a change of view state appears on the screen. You define a transition by using a combination of the Flex effect classes. For more information on transitions, see “Using Transitions” on page 885 in *Adobe Flex 3 Developer Guide*.

You can define your own custom effects for use in transitions. To do so, you have to account for the effect being used in a transition when you override the `EffectInstance.play()` method. The `EffectInstance.play()` method must be able to determine default values for effect properties when the effect is used in a transition.

Defining the default values for a transition effect

Like any effect, an effect in 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.

Flex uses the following rules to determine the start and end values of effect properties when you use the effect in a transition:

1 If the effect defines the values of any properties, it uses the properties 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 define the properties of the effect.

2 If the effect does not define the start values of the effect, the effect determines the values from the `EffectInstance.propertyChanges` property passed to the effect instance. Flex sets the `propertyChanges` property by using information from the current settings of the component, as defined by the current view state. For more information on the `propertyChanges` property, see "How Flex initializes the propertyChanges property" on page 213.

In the example in step 1, notice that the Move and Resize effects do not define start values. Therefore, Flex determines the start values from the current size and position of the effect targets in the current view state, and passes that information to each effect instance by using the `propertyChanges` property.

3 If the effect does not define the end values of the effect, the effect determines the values from the `Effectinstance.propertyChanges` property passed to the effect instance. Flex sets the `propertyChanges` property by using information about the component, as defined by the destination view state. For more information on the `propertyChanges` property, see "How Flex initializes the propertyChanges property" on page 213.

In the example in rule 1, Flex determines the end values of the Move and Resize effects from the size and position of the effect targets in the destination view state. In some cases, the destination view state defines those values. If the destination view state does not define the values, Flex determines them from the setting of the base view state, and passes that information to each effect instance by using the `propertyChanges` property.

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.

Using the propertyChanges property

The `EffectInstance.propertyChanges` property contains a PropertyChanges object. A PropertyChanges object contains the properties described in the following table:

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 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 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}`

In the body of the `Effectinstance.play()` method, you can examine the information in the `propertyChanges` property to configure the effect.

How Flex initializes the propertyChanges property

Before you can use the `EffectInstance.propertyChanges` property in your effect instance, it has to be properly initialized. When you change the view state, Flex initializes `propertyChanges.start` and `propertyChanges.end` separately.

The following steps describe the actions that occur when you change view states. Notice that Flex initializes `propertyChanges.start` as part of step 4, and initializes `propertyChanges.end` as part of step 7.

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 view state.
4. Flex calls the `Effect.captureStartValues()` method to initialize `propertyChanges.start` for all effect instances.

   You can also call `Effect.captureStartValues()` to initialize an effect instance used outside of a transition.
5. Flex applies the destination view state to the application.
6. Flex dispatches the `currentStateChange` event.
7. Flex plays the effects defined in the transition.

   As part of playing the effect, Flex invokes the factory class play() method and the `Effect.play()` method to initialize `propertyChanges.end` for effect instances.

Example: Creating a transition effect

The following example modifies the Rotation effect created in “Example: Creating a tween effect” on page 208 to make the effect usable in a transition.

This example shows the RotationTrans.as class that defines the factory class for the effect. To create RotationTrans.as, the only modification you make to Rotation.as is to remove the default value definitions for the `angleFrom` and `angleTo` properties. The `RotationTransInstance.play()` method determines the default values.

```
package myEffects
{
   // myEffects/RotationTrans.as
   import mx.effects.TweenEffect;
   import mx.effects.EffectInstance;
   import mx.effects.IEffectInstance;
```

```
    public class RotationTrans extends TweenEffect
    {
        // Define parameters for the effect.
        // Do not specify any default values.
        // The default value of these properties is NaN.
        public var angleFrom:Number;
        public var angleTo:Number;

        // Define constructor with optional argument.
        public function RotationTrans(targetObj:Object = null) {
            super(targetObj);
            instanceClass= RotationTransInstance;
        }

        // Override getAffectedProperties() method to return "rotation".
        override public function getAffectedProperties():Array {
            return ["rotation"];
        }

        // Override initInstance() method.
        override protected function initInstance(inst:IEffectInstance):void {
            super.initInstance(inst);
            RotationTransInstance(inst).angleFrom = angleFrom;
            RotationTransInstance(inst).angleTo = angleTo;
        }
    }
}
```

In the RotationTransInstance.as class, you modify the `play()` method to calculate the default values for the `angleFrom` and `angleTo` properties. This method performs the following actions:

1. Determines whether the user set values for the `angleFrom` and `angleTo` properties.
2. If not, determines whether the `Effectinstance.propertyChanges` property was initialized with start and end values. If so, the method uses those values to configure the effect.
3. If not, sets the `angleFrom` and `angleTo` properties to the default values of 0 for the `angleFrom` property, and 360 for the `angleTo` property.

The following example shows the RotationTransInstance.as class:

```
package myEffects
{
    // myEffects/RotationTransInstance.as
    import mx.effects.effectClasses.TweenEffectInstance;
    import mx.effects.Tween;
```

```
public class RotationTransInstance extends TweenEffectInstance
{
    // Define parameters for the effect.
    public var angleFrom:Number;
    public var angleTo:Number;

    public function RotationTransInstance(targetObj:Object) {
        super(targetObj);
    }

    // Override play() method class.
    override public function play():void {
        // All classes must call super.play().
        super.play();

        // Check whether angleFrom is set.
        if (isNaN(angleFrom))
        {
          // If not, look in propertyChanges.start for a value.
          // Otherwise, set it to 0.
          angleFrom = (propertyChanges.start["rotation"] != undefined) ?
                propertyChanges.start["rotation"] : 0;

        }

        // Check whether angleTo is set.
        if (isNaN(angleTo))
        {
            // If not, look in propertyChanges.end for a value.
            // Otherwise, set it to 360.
            angleTo = (propertyChanges.end["rotation"] != undefined) ?
                propertyChanges.end["rotation"] : 360;
        }

        // Create a Tween object. The tween begins playing immediately.
        var tween:Tween =
            createTween(this, angleFrom, angleTo, duration);
    }

    // Override onTweenUpdate() method.
    override public function onTweenUpdate(val:Object):void {
        target.rotation = val;
    }

    // Override onTweenEnd() method.
    override public function onTweenEnd(val:Object):void {
            // All classes that implement onTweenEnd()
            // must call super.onTweenEnd().
            super.onTweenEnd(val);
        }
    }
}
```

The following application uses the RotationTrans effect:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- effects/MainRotationTrans.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myEffects.*">

    <!-- 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:Parallel>
```

```
                    <mx:Move  duration="400"/>
                    <mx:Resize duration="400"/>
                </mx:Parallel>
                <MyComp:RotationTrans filter="move"/>
            </mx:Sequence>
        </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>
```

Creating a custom data effect

You can create a custom data effect for use with the List and TileList controls. To create a custom data effect, define a Parallel or Sequence effect, and then assign the effect to the `itemsChangeEffect` property of the control.

Flex defines several action effects so that you can control when items are added, removed, or replaced in the list control as part of the effect, or to allow you to control the layout of item renderers in the control during the effect. The following table describes these action effects:

Action effect class	**Use**
AddItemAction	Defines an effect that determines when the item renderer appears in the control for an item being added to a list-based control, or for an item that replaces an existing item in the control.
RemoveItemAction	Defines an effect that determines when the item renderer disappears from the control for the item renderer of an item being removed from a list-based control, or for an item that is replaced by a new item added to the control.
UnconstrainItemAction	Defines an effect to temporarily stop item renderers from being positioned by the layout algorithm of the parent control. Use this effect to allow item renderers in a TileList control to move freely rather than being constrained to lay in the normal grid that is defined by the control. You typically add this effect when your custom data effect moves item renderers.

You can use the `Effect.filter` property to filter possible effect targets, which lets you play an effect only on a subset of the item renderers in the list-based control. The following table lists the values of the `Effect.filter` property that you use with data effects:

Effect.filter value	**Descriptions**
`addItem`	Play the effect on the item renderer for any list items added to a List or TileList control.
`removeItem`	Play the effect on the item renderer for any list items removed from a List or TileList control.
`replacedItem`	Play the effect on the item renderer for any list items replaced in a List or TileList control by a new item.
`replacementItem`	Play the effect on the item renderer for any list items added to a List or TileList control that replaces an existing item.

For detailed information and examples of action effects and effect filters, see “Using action effects in a transition” on page 895 and “Filtering effects” on page 898.

Example: Custom data effect

The following example defines a custom data effect for the TileList control. This data effect defines a Sequence effect that contains a Blur effect and a Parallel effect. The data effect performs the following actions when you delete an item from the TileList control:

- Applies the Blur effect to the item being deleted.
- Makes the item being deleted invisible.

- Applies the UnconstrainItemAction effect because you are moving item renderers in the custom effect.
- Moves the remaining items to their new position.
- Removes the item being deleted.

The custom data effect performs the following actions when you add an item to the TileList control:

- Applies the UnconstrainItemAction effect because you are moving item renderers in the custom effect.
- Moves the existing items to their new position.
- Adds the new item to the control.
- Applies a Blur effect to the new item.

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

    <mx:Script>
        <![CDATA[
            import mx.effects.easing.Elastic;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection(
                ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q']);

            private function deleteItem():void {
                // As each item is removed, the index of the other items changes.
                // So first get the items to delete, and then determine their indices
                // as you remove them.
                var toRemove:Array = [];
                for (var i:int = 0; i < tlist0.selectedItems.length; i++)
                    toRemove.push(tlist0.selectedItems[i]);
                for (i = 0; i < toRemove.length; i++)
                    myDP.removeItemAt(myDP.getItemIndex(toRemove[i]));
            }

            private var zcount:int = 0;
            private function addItem():void {
                // Always add the new item after the third item,
                // or after the last item if the length is less than 3.
                myDP.addItemAt("Z"+zcount++,Math.min(3,myDP.length));
            }
        ]]>
    </mx:Script>

    <!-- Define a custom data effect as a Sequence effect. -->
```

```
<mx:Sequence id="dataChangeEffect1">
    <mx:Blur
        blurYTo="12" blurXTo="12"
        duration="300"
        perElementOffset="150"
        filter="removeItem"/>
    <mx:SetPropertyAction
        name="visible" value="false"
        filter="removeItem"/>
    <mx:UnconstrainItemAction/>
    <mx:Parallel>
        <mx:Move
            duration="750"
            easingFunction="{Elastic.easeOut}"
            perElementOffset="20"/>
        <mx:RemoveItemAction
            startDelay="400"
            filter="removeItem"/>
        <mx:AddItemAction
            startDelay="400"
            filter="addItem"/>
        <mx:Blur
            startDelay="410"
            blurXFrom="18" blurYFrom="18" blurXTo="0" blurYTo="0"
            duration="300"
            filter="addItem"/>
    </mx:Parallel>
</mx:Sequence>

<!-- This TileList uses a custom data change effect. -->
<mx:TileList id="tlist0"
    height="400" width="400"
    fontSize="30" fontStyle="bold"
    columnCount="4" rowCount="4"
    direction="horizontal"
    dataProvider="{myDP}"
    allowMultipleSelection="true"
    offscreenExtraRowsOrColumns="4"
    itemsChangeEffect="{dataChangeEffect1}"/>

<mx:Button
    label="Delete selected item(s)"
    click="deleteItem();"/>
<mx:Button
    label="Add item"
    click="addItem();"/>
```

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

This example uses the `Effect.filter` property to filter possible effect targets. Because you set the `filter` property of the first Blur effect to `removeItem`, Flex only applies the effect to items added to the list control. Because the Move effect omits the `filter` property, Flex applies the effect to all items in the control.

You can create a custom filter by passing an instance of the EffectTargetFiler class to the `Effect.customFilter` property. For example, the `Effect.filter` property lets you specify only a single value, such as `addItem` or `removeItem`. However, you can use the `EffectTargetFiler.requiredSemantics` property to specify a collection of properties and associated values that must be associated with a target for the effect to be played. For more information, see “Filtering effects” on page 898 in *Adobe Flex 3 Developer Guide*.

Debugging a custom data effect

When debugging a custom data effect, you should set the `Effect.suspendBackgroundProcessing` property to `true`. If `true`, the `suspendBackgroundProcessing` property blocks all background processing while an effect is playing, and can help to reduce any flickering caused by the effect. Background processing includes measurement, layout, and processing responses that have arrived from the server.

Creating a custom data effect in a separate MXML file

You can write your custom data effect as an MXML component, as the following example shows:

```
<?xml version="1.0"?>
<!-- dataEffects\myComponents\MyDataEffect.mxml -->
<mx:Sequence xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.effects.easing.Elastic;
        ]]>
    </mx:Script>

    <!-- Define a custom data effect as a Sequence effect. -->
    <mx:children>
        <mx:Blur
            blurYTo="12" blurXTo="12"
            duration="300"
            perElementOffset="150"
            filter="removeItem"/>
        <mx:SetPropertyAction
            name="visible" value="false"
            filter="removeItem"/>
        <mx:UnconstrainItemAction/>
        <mx:Parallel>
            <mx:Move
                duration="750"
                easingFunction="{Elastic.easeOut}"
                perElementOffset="20"/>
            <mx:RemoveItemAction
                startDelay="400"
                filter="removeItem"/>
            <mx:AddItemAction
                startDelay="400"
                filter="addItem"/>
            <mx:Blur
                startDelay="410"
                blurXFrom="18" blurYFrom="18" blurXTo="0" blurYTo="0"
                duration="300"
                filter="addItem"/>
        </mx:Parallel>
    </mx:children>
</mx:Sequence>
```

In this example, you create a custom MXML component based on the Sequence effect. Notice that the component specifies the `Sequence.children` property, the default MXML property of the Sequence effect class. This tag is necessary in a custom component because custom component do not recognize the default MXML property; therefore, you must specify it. For more information on the default MXML property, see “MXML Syntax” on page 23.

The custom MXML component is in the file MyDataEffect.mxml in the myComponents subdirectory of the main application file. The following example shows the main application that uses this MXML component:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import myComponents.MyDataEffect;

            [Bindable]
            private var myDP:ArrayCollection = new ArrayCollection(
                ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q']);

            private function deleteItem():void {
                // As each item is removed, the index of the other items changes.
                // So first get the items to delete, then determine their indices
                // as you remove them.
                var toRemove:Array = [];
                for (var i:int = 0; i < tlist0.selectedItems.length; i++)
                    toRemove.push(tlist0.selectedItems[i]);
                for (i = 0; i < toRemove.length; i++)
                    myDP.removeItemAt(myDP.getItemIndex(toRemove[i]));
            }

            private var zcount:int = 0;
            private function addItem():void {
                // Always add the new item after the third item,
                // or after the last item if the length is less than 3.
                myDP.addItemAt("Z"+zcount++,Math.min(3,myDP.length));
            }
        ]]>
    </mx:Script>

    <!-- This TileList uses a custom data change effect -->
    <mx:TileList id="tlist0"
        height="400" width="400"
        fontSize="30" fontStyle="bold"
        columnCount="4" rowCount="4"
        direction="horizontal"
        dataProvider="{myDP}"
        allowMultipleSelection="true"
        offscreenExtraRowsOrColumns="4"
        itemsChangeEffect="{MyDataEffect}"/>

    <mx:Button
        label="Delete selected item(s)"
        click="deleteItem();"/>
    <mx:Button
        label="Add item"
        click="addItem();"/>
</mx:Application>
```

Defining a custom effect trigger

You can create a custom effect trigger to handle situations for which the standard Flex triggers do not meet your needs. An effect trigger is paired with a corresponding event that invokes the trigger. For example, a Button control has a `mouseDown` event and a `mouseDownEffect` trigger. The event initiates the corresponding effect trigger when a user clicks a component. You use the `mouseDown` event to specify the event listener that executes when the user selects the component. You use the `mouseDownEffect` trigger to associate an effect with the trigger.

Suppose that you want to apply an effect that sets the brightness level of a component when a user action occurs. The following example shows a custom Button control that uses a new property, `bright`, and dispatches two new events, `darken` and `brighten`, based on changes to the `bright` property. The control also defines two new effect triggers, `darkenEffect` and `brightenEffect`, which are paired with the `darken` event and the `brighten` event.

```
<?xml version="1.0"?>
<!-- effects\myComponents\MyButton.mxml -->
<mx:Button xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Metadata>
        <!-- Define the metadata for the events and effect triggers. -->
        [Event(name="darken", type="flash.events.Event")]
        [Event(name="brighten", type="flash.events.Event")]
        [Effect(name="darkenEffect", event="darken")]
        [Effect(name="brightenEffect", event="brighten")]
    </mx:Metadata>

    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            // Define the private variable for the bright setting.
            private var _bright:Boolean = true;

            // Define the setter to dispatch the events
            // corresponding to the effect triggers.
            public function set bright(value:Boolean):void {
                _bright = value;

                if (_bright)
                    dispatchEvent(new Event("brighten"));
                else
                    dispatchEvent(new Event("darken"));
            }

            // Define the getter to return the current bright setting.
            public function get bright():Boolean {
                return _bright;
            }
        ]]>
    </mx:Script>

</mx:Button>
```

When you declare an event in the form `[Event(name="`*`eventName`*`", type="`*`package.eventType`*`")]`, you can also create a corresponding effect, in the form `[Effect(name="`*`eventname`*`Effect", event="`*`eventname`*`")]`. As in the previous example, in the `<mx:Metadata>` tag, you insert the metadata statements that define the two new events, `darken` and `brighten`, and the new effect triggers, `darkenEffect` and `brightenEffect`, to the Flex compiler.

For more information on using metadata, see "Metadata Tags in Custom Components" on page 33.

The application in the following example uses the MyButton control. The `darkenEffect` and `brightenEffect` properties are set to the FadeOut and FadeIn effects, respectively. The `click` event of a second Button control toggles the MyButton control's `bright` property and executes the corresponding effect (FadeOut or FadeIn).

```
<?xml version="1.0"?>
<!-- effects/MainMyButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <!-- Define two fade effects for darkening and brightening target. -->
    <mx:Fade id="FadeOut" duration="1000" alphaFrom="1.00" alphaTo=".20"/>
    <mx:Fade id="FadeIn" duration="1000" alphaFrom=".20" alphaTo="1.00"/>

    <!-- Define custom button that defines the
        darkenEffect and brightenEffect. -->
    <MyComp:MyButton
        label="MyButton" id="btn"
        darkenEffect="{FadeOut}"
        brightenEffect="{FadeIn}"
        darken="debugW.text='got darken event';"
        brighten="debugW.text='got brighten event';"/>

    <!-- Define button that triggers darken event. -->
    <mx:Button
        label="set bright to false"
        click="btn.bright = false; myTA.text=String(btn.bright);"/>

    <!-- Define button that triggers brighten event. -->
    <mx:Button
        label="set bright to true"
        click="btn.bright = true; myTA.text=String(btn.bright);"/>

    <!-- TextArea displays the current value of bright. -->
    <mx:TextArea id="myTA" />

    <!-- TextArea displays event messages. -->
    <mx:TextArea id="debugW" />
    <!-- Define button to make sure effects working. -->
    <MyComp:MyButton id="btn2" label="test effects"
        mouseDownEffect="{FadeOut}"
        mouseUpEffect="{FadeIn}"/>

</mx:Application>
```

Overriding the initEffect() method

The EffectInstance class defines the `initEffect()` method that you can override in your custom effect. This method has the following signature:

```
public initEffect(event:Event):void
```

where *`event`* is the Event object dispatched by the event that triggered the effect.

For example, a user might create an instance of an effect, but not provide all of the configuration information that is required to play the effect. Although you might be able to assign default values to all properties within the definition of the effect class, in some cases you might have to determine the default values at run time.

In this method, you can examine the event object and the effect target to calculate values at run time. For more information on how to create a custom event and an effect trigger, see "Defining a custom effect trigger" on page 223. As part of that example, you can add properties to the event object passed to the `dispatchEvent()` method. You can then access that event object, and its additional properties, from the `initEffect()` method.

By overriding the `initEffect()` method, you can also access the `target` property of the Event object to reference the target component of the effect. For example, if you must determine the current x and y coordinates of the component, or its current height and width, you can access them from your override of the `initEffect()` method.