

# CHAPTER 10

## Components: Designing for Reusability

### IN THIS CHAPTER

Building Applications with Flex Components 225

Extending the Flex Components 225

Creating New Components 241

### Building Applications with Flex Components

Components are the building blocks of all Flex applications—from navigator containers, which help guide the user through different child components, to layout containers, which control the size and positioning of their children, to the controls that make up a user interface. They define what’s seen by the user and how the user can interact with the application.

In Chapters 5–7, we discussed the containers and controls that ship with Flex, which covers the broad range of components you would expect to have available in your toolbox as a developer.

However, not all applications are the same; at some point, you’ll want something a little bit different from what ships with Flex, such as the persistent ComboBox we introduced in Chapter 7, “Using Form Controls.” There may be times when you want something completely new—a container or control that introduces features not available in any of the components that ship with Flex. You may even want to create your own custom application component to encapsulate a framework that all your applications use. Thankfully, Flex gives us hooks into its component architecture for the building of such custom components.

This chapter covers ways to create your own custom components, by using both MXML and ActionScript. We discuss the various options, and when and why you would go down a certain path. Finally, we touch on how components can be built outside of the Flex environment by using Flash MX 2004 to produce packaged components that can then be reused by your Flex application.

### Extending the Flex Components

Flex’s architecture makes it easy to build custom components and provides two different ways to do so: either via pure MXML or through creating an ActionScript 2 class. Which strategy to use is entirely up to the developer, but our recommendation is that for anything other than the simplest

extension to a built-in component, you're better off using ActionScript. Although creating custom components in MXML is in many ways a faster process, you lose the flexibility of a class-based development environment in which you can encapsulate your functionality into multiple classes, increasing reusability across your custom components, and allowing you to test them using unit testing frameworks. MXML-only components lead you down the path of including everything in a single file, resulting in a more convoluted and unreadable solution, with minimal reusability.

Of course, you also can mix the two solutions, creating an MXML component that utilizes your ActionScript classes.

Let's look at examples of each type of solution by extending the Flex ComboBox control.

## MXML Components

You can create MXML components to extend a built-in Flex component, adding specific functionality as necessary. In this example, we want to provide a `<CountryComboBox>` control that a developer can use in MXML pages to display a ComboBox with a list of predefined countries. This is an example of creating a component for reuse.

The MXML resides in a file given the same name as your component name. So when creating the `<CountryComboBox>` component, we save the MXML in a file called `CountryComboBox.mxml`.

The root tag of a MXML component is the parent tag; because we're extending the ComboBox, our custom component MXML is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:dataProvider>
    <mx:Array>
      <mx:String>Scotland</mx:String>
      <mx:String>England</mx:String>
      <mx:String>N. Ireland</mx:String>
      <mx:String>Wales</mx:String>
    </mx:Array>
  </mx:dataProvider>
</mx:ComboBox>
```

And that's it—we created a custom component that can be used in any of our MXML applications.

We show here how it's used:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

  <CountryComboBox xmlns="*" />

</mx:Application>
```

**Figure 10.1** shows how the application with the custom `<CountryComboBox>` looks to the user.

You may have noticed that our `<CountryComboBox>` component has a namespace definition, as shown here:

```
<i2:CountryComboBox xmlns:i2="*" />
```

**Figure 10.1**

The `<CountryComboBox>` custom component used within an application.



This specifies that the component is in a namespace called `i2` and tells Flex that the component can be found in the same directory as the MXML file. Later in this chapter, when we create an ActionScript component, you'll see how we can use the namespace to tell Flex where it can find our classes in a package hierarchy.

### Namespaces

Using namespaces lets you refer to more than one set of XML tags within a single XML document. Flex uses the namespaces of custom components to locate those components in its classpath.

### Inherited Methods and Properties

In the previous example, we defined the instance of our `<CountryComboBox>` without any additional properties. However, because our custom control extends the Flex `ComboBox`, it automatically inherits all the methods and properties of the base component. So, in our usage of the custom component, we can perform the same actions we can do to a standard `ComboBox`, as in this example:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <CountryComboBox id="theComboBox" change="comboChanged" xmlns="*" />

</mx:Application>
```

We have now given our control an `id` property, and attached a `change` handler to it so that the function `comboChanged()` will be called when the user selects a new item from the list. Because we've extended the standard Flex `ComboBox`, you can use any of its properties or methods this way.

### Adding Custom Methods and Properties

As well as overriding existing functionality, as in the preceding example, in which we set the items in the `ComboBox`, when we define custom components we can define our own properties and methods for the component.

In Chapter 7, we showed how we could create a `ComboBox` that persisted user additions to a local shared object, which were included in the drop-down list and included again on the next display of the control. In that example, we included all the custom code in our main application MXML file, precluding us from reusing that component elsewhere—we'd have to take a copy of the code,

which leads to maintenance problems if the control is to be enhanced. Instead, we'll mix that functionality with the `<CountryComboBox>` we created previously to produce a new custom component, `<EditableCountryComboBox>`.

This `<EditableCountryComboBox>` will have a predefined list of countries available for the user to select from, but they will also be able to add their own entries to the list, which can be persisted to a local shared object. By default, the control won't persist user additions across invocations of the application; the developer of the application using our component will decide, via a property of our control, whether user entries should be persisted.

The component is defined as follows, in the file `EditableCountryComboBox.mxml`, with the highlighted code being of particular interest with regard to custom components:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml"
              editable="true" enter="addItemToCombo( event );"
              initialize="addSharedObjectItems( this );" >

  <mx:Boolean id="persistUserAdditions">false</mx:Boolean>

  <mx:Script>

  <![CDATA[
private function addItemToCombo( event )
{
  if ( text.length > 0 && !itemExistsInComboDataProvider( text ) )
  {
    addItem(text );

    if ( persistUserAdditions )
    {
      var comboBoxSharedObject = SharedObject.getLocal( id );
      comboBoxSharedObject.data.dataProvider = dataProvider;
      comboBoxSharedObject.flush();
    }
  }
}

private function addSharedObjectItems( combo )
{
  var comboBoxSharedObject = SharedObject.getLocal( id );
  if ( !persistUserAdditions )
  {
    delete comboBoxSharedObject.data.dataProvider;
    return;
  }
  var sharedObjectDataProvider = comboBoxSharedObject.data.dataProvider;

  for ( var p in sharedObjectDataProvider )
  {
    if ( !itemExistsInComboDataProvider( sharedObjectDataProvider[p] ) )
    {
      dataProvider.addItem( sharedObjectDataProvider[p] );
    }
  }
}
```

```
    }  
  
    private function itemExistsInComboDataProvider( combo, item )  
    {  
        for ( var p in combo.dataProvider )  
        {  
            if ( item == combo.dataProvider[p] )  
                return true;  
        }  
        return false;  
    }  
}]]>  
</mx:Script>  
  
<mx:dataProvider>  
    <mx:Array>  
        <mx:String>Scotland</mx:String>  
        <mx:String>England</mx:String>  
        <mx:String>N. Ireland</mx:String>  
        <mx:String>Wales</mx:String>  
    </mx:Array>  
</mx:dataProvider>  
  
</mx:ComboBox>
```

Once again, the root tag of the custom component is `<mx:ComboBox>`, meaning that we pick up all the properties and methods of that Flex control. The remainder of the code is very similar to that shown in Chapter 7, with a few additions and changes.

In the `<mx:ComboBox>` root tag definition, we set the `editable` property to `true` and attach handlers to the `enter` event, which is broadcast by the base class `ComboBox` control when the user presses the Enter key when focused in the editable part of the control, and the `initialize` event, which is broadcast by Flex during the instantiation of our component and where we add any previously added items from our local shared object.

#### initialize **Event**

The `initialize` event is broadcast by the `mx.core.UIObject` class, which is the base class for all Flex components.

In the creation and retrieval of the local shared object, we now use the `id` property of the control, rather than the fixed `String` we used in Chapter 7. This is to ensure that multiple instances of the control keep track of their own list of user-added items. This means that two instances of the control in different MXML files (but with the same `id`) will share user items. This was our choice—we could have obfuscated our local shared object identifier further if we wanted to ensure complete uniqueness across MXML files. We also could have created a new property, to be set by the developer using the component, and used that as the local shared object identifier.

Finally, we introduced the `persistUserAdditions` property. This property is defined via the `<mx:Boolean>` element, but we could have defined it in our `<mx:Script>` block using a standard ActionScript variable definition. We defaulted its value to `false`, meaning that user entries will not be persisted unless told otherwise.

Here's how we use our new component to persist user additions to the list of countries in the `ComboBox`:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <EditableCountryComboBox id="theEditableComboBox"
                            persistUserAdditions="true" xmlns="*" />

</mx:Application>
```

We set the `id` and `persistUserAdditions` properties of the new component. Now, when the user adds entries and restarts the application, the user-added items will be included in the `ComboBox`.

`<EditableCountryComboBox>` **and** `id`

If we don't give our `<EditableCountryComboBox>` an `id` property, Flex will allocate one. Although doing this allows the component to work okay and still persist the user additions to the list of items in the `ComboBox`, there's no guarantee that Flex will allocate the same `id` the next time the component is compiled—in which case the user additions won't be shown on next invocation. More importantly, Flex may choose an `id` that was previously used for a completely different instance of our custom control, such as to show a list of job titles. They would then be retrieved and shown in our country list! For that reason, we recommend defining an `id` property with the control.

Our `<EditableCountryComboBox>` is now working as we want it. Developers can drop it into any of their applications and use the standard `ComboBox` features such as handling events when the user selects an item from the list, or our custom feature of persisting the user entries.

However, there are various drawbacks to what we've produced:

- Our component is editable by default. What if we want to use a `ComboBox` with a list of countries, but not let the user edit it or add list items? We'd have to duplicate the component and change it to make it noneditable. Thereafter, if we wanted to add another country to our application, we'd have to make the change to each component, leading to increased maintenance effort.
- Our component can deal with a list of countries only. What if we now want to allow users to select a job title from a list or add a job title if it doesn't exist? With our current setup, we'd have to duplicate the MXML file we created previously. What if we then wanted to add a new feature to our editable `ComboBoxes`? Again, we'd have to make the same change to every copy we'd made.

- User entries are always added to the drop-down portion of the control. What if we want to change this setup at runtime, so that sometimes new entries are added and other times they're not? Once again, we'd have to duplicate the existing code.
- The MXML file contains a lot of code and little actual MXML content. MXML files should contain primarily MXML tags. Having so much code in the file itself could lead to maintenance issues in the future as new features are added to the control.

So, to alleviate these problems, we add to our component, at the same time converting it from one defined in an MXML file to one defined in an ActionScript class.

## ActionScript Components

ActionScript components, as their name suggests, are defined in ActionScript classes instead of in an MXML file. However, to the end user, there's no difference—the component is available in the user's applications. Indeed, a developer using a custom component shouldn't know (nor should he care) whether that component is built-in MXML or ActionScript.

Let's use the `<EditableCountryComboBox>` we created previously to create a new component, changing how it works as follows:

- Implement it in an ActionScript class.
- Remove the default behavior of the component being editable.
- Remove the hard-coded list of countries in the control, instead making it a generic editable combo box.
- Add an `addUserItemsToList` property to the component, which dictates whether user additions are added to the drop-down list in the control (this is separate from whether those additions are persisted to a local shared object).

### Class Definition

First of all, we define our class. We're extending the Flex `ComboBox` component, so our class definition is as follows, in a file called `EditableComboBox.as`:

```
class com.iterationtwo.components.EditableComboBox extends
    mx.controls.ComboBox
{
    ...
}
```

The Flex `ComboBox` component is defined in the class `mx.controls.ComboBox`, so our class extends that class to inherit all its properties and methods.

#### `EditableComboBox` Class Package

We created our `EditableComboBox` custom component in the package `com.iterationtwo.components`, meaning that it must reside in the `com/iterationtwo/components` directory structure when used. We'll show later in this chapter how we use namespaces to specify where the component can be found.

## Instance Properties

Our custom component has two properties over and above those it inherits from the standard Flex `ComboBox`, which are defined as follows:

```
public var persistUserList:Boolean = false;
public var addUserItemsToList:Boolean = true;
```

The `persistUserList` property has the same function as in the `MXML` component we created previously—it's used to determine whether user additions should be persisted to a local shared object.

The new addition is the `addUserItemsToList` property, which determines whether the user can actually add new items to the drop-down list.

## Component Events

Before visiting our class constructor, we'll first explain a bit about the events that are triggered during the construction of Flex components.

As a component is constructed, Flex broadcasts the following events:

1. The `initialize` event is broadcast when the component and all its children (if it has any) have been instantiated. Flex always guarantees that the children of a component have been instantiated by the time it broadcasts the parent's `initialize` event.
2. The `load` event is broadcast when the component has been loaded into Flash Player.
3. The `draw` event is broadcast when the component has been drawn internally in the player, but not yet drawn to the user interface.
4. The `creationComplete` event is broadcast when the component has been measured, laid out, and drawn, but not yet shown on the user interface. This is the final hook you can intercept before the component is shown to the user.
5. The `show` event is broadcast when the component has been shown on the user interface. Hidden components don't fire this event.

We can use these events as hooks into a component's construction, acting on them as necessary, by adding event listeners to the events using `ActionScript` in the following form:

```
addEventListener( "eventName", myHandler );
```

With this code in place, when the event `eventName` is broadcast, the method `myHandler` is called with the event object for that event. This is the equivalent to handling the event in `MXML` using the usual format:

```
<CustomControl eventName="myHandler( event );" />
```

We'll use the `addEventListener()` function in our component constructor.

## Constructors

The constructor of the custom component is called by Flex as it creates the user interface at runtime, and is defined as follows:

```
public function EditableComboBox()
{
    super();
    addEventListener( "initialize", myInitializeHandler );
    addEventListener( "enter", myEnterHandler );
}
```

First of all, we call `super()` so that the Flex `ComboBox` constructor is called; we then add listeners to two events.

We add a listener to the `initialize` event of the control being constructed, which we described previously. Flex broadcasts that event after the component and its children have been instantiated. When the event is broadcast, our `myInitializeHandler()` method is called.

Similarly, we add a listener to the `enter` event, which is broadcast by the Flex `ComboBox` when the user hits the Enter key in the control. In this instance, we call the `myEnterHandler()` method of our class.

The previous way is recommended to attach handlers to events in a custom component.

## Event Handlers

Our two handlers are defined as follows:

```
private function myInitializeHandler( event )
{
    addSharedObjectItems();
}

public function myEnterHandler( event )
{
    addItemToCombo( event );
}
```

When the `initialize` event is received and the `myInitializeHandler()` method called, we retrieve the items stored in the local shared object and, depending on our `persistUserList` and `addUserItemsToList` properties, add them to the list via the `addSharedObjectItems()` method.

When the `enter` event is received and the `myEnterHandler()` method called, we call the `addItemToCombo()` method to add the user entry to the drop-down portion of the `ComboBox`.

The definitions of the `addSharedObjectItems()` and `addItemToCombo()` methods are the same as in the `MXML` component we created previously.

And that's the complete `ActionScript` version of the component. You can see the full listing later on, but not before we show you how our new component can be used.

## Using the Component

Because we removed the `dataProvider` property from our component definition, it can now be passed from the application using our component. This has made our custom component completely generic, meaning that it can be used for choosing from any list of items, as shown here, where we show a predefined list of job titles the user can add to.

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
                xmlns:i2="com.iterationtwo.components" >

    <i2:EditableComboBox id="jobTitleComboBox"
        editable="true"
        persistUserList="true"
        addUserItemsToList="true" >
        <i2:dataProvider>
            <mx:Array>
                <mx:String>J2EE Developer</mx:String>
                <mx:String>.NET Developer</mx:String>
                <mx:String>Flash Designer</mx:String>
                <mx:String>Graphic Designer</mx:String>
            </mx:Array>
        </i2:dataProvider>
    </i2:EditableComboBox>

</mx:Application>

```

Remember that we didn't default the `editable` property of the `ComboBox` to `true` in our custom component definition, meaning that developers can use our component as a straight swap in place of the standard Flex component, and have it act the same way as that component if required.

So, we set the `editable` property to `true` in the instance definition in our MXML file; we also set the `persistUserList` and `addUserItemsToList` properties to the same value. By setting any of these properties to `false`, you can change the behavior of the component.

Finally, note that we defined the `i2` namespace in the MXML file. By giving that namespace the value of `com.iterationtwo.components` and by using that namespace on our `<i2:EditableComboBox>` instance definition, we're telling Flex where it can find the component in the `com/iterationtwo/components` directory structure in its classpath.

There's still one subtle shortcoming of the component we created. We're allowing the user to add entries to the drop-down list of the combo box, which can be persisted to a local shared object. However, what if the developer of the application using the control wants to perform some other action when the user adds his or her own entry to the list? Enterprise companies may want to monitor the job titles people are adding to their predefined list or capture the reason for a loan to increase up-selling and cross-selling opportunities. With the component we created, there's no way for the developer to be notified when the user adds an entry. We can fix this problem by building in an event to our custom component.

## Broadcasting Events

Flex uses an event-driven model to notify other parts of the application that an action has taken place. We can hook into this model by adding events to our own components.

When we created our custom `ComboBox` in which the user could add items to the drop-down portion of the control, we noted that developers using that control may want to be notified when the user adds an entry. This can be achieved very easily with Flex.

All Flex components extend, in their hierarchical path, the `mx.core.UIObject` class, which contains a `dispatchEvent()` method with the following signature:

```
dispatchEvent( eventObject );
```

The `eventObject` can contain any number of properties, but should contain the `type` property, which is the name of the event being broadcast.

We also want to add a property giving the text added to the `ComboBox` by the user. So, in our custom component, we add the following line after the user's entry has been added to the control:

```
dispatchEvent( { type: "userItemAdded" } );
```

This tells Flex to broadcast an event called `userItemAdded`, with three properties: `type`, `target`, and `item`.

It would be easy to think that the above is all that's required—we now just need to handle the `userItemAdded` event in our main application to act on the user addition. However, if we try to attach a handler to that event in our application MXML, we get the following error from the Flex compiler:

```
Error TestEditableComboBox.mxml:8 unknown attribute 'userItemAdded' on
com.iterationtwo.components.EditableComboBox
```

This error is returned because we haven't informed the Flex compiler that our new component broadcasts the `userItemAdded` event. To do this, we add this Event metadata to the top of the component class.

```
[Event("userItemAdded")]
```

This line tells the Flex compiler that our component can broadcast the `userItemAdded` event, and lets applications using the component attach a handler for that event. The Metadata component is covered in more detail in Chapter 15, "ActionScript 2.0 and Flex."

### Metadata in MXML

Metadata can also be defined in an MXML file, using the `<mx:Metadata>` tag. If we wanted to implement the event broadcasting in our MXML custom component, we would include the following in our MXML file:

```
<mx:Metadata>
  [Event("userItemAdded")]
</mx:Metadata>
```

### Component Listing

Here's the full listing of the custom component defined as an ActionScript class in the file `EditableComboBox.as`.

```
[Event("userItemAdded")]

class com.iterationtwo.components.EditableComboBox extends
  mx.controls.ComboBox
{
  public var persistUserList:Boolean = false;
  public var addUserItemsToList:Boolean = true;
```

```

function EditableComboBox()
{
  super();
  addEventListener( "initialize", myInitializeHandler );
  addEventListener( "enter", myEnterHandler );
}

private function myInitializeHandler( event )
{
  addSharedObjectItems();
}

public function myEnterHandler( event )
{
  addItemToCombo( event );
}

private function addItemToCombo( event )
{
  {
    if ( text.length > 0 && addUserItemsToList )
    {
      if ( !itemExistsInComboDataProvider( text ) )
      {
        addItem( text );

        if ( persistUserList )
        {
          var comboBoxSharedObject = SharedObject.getLocal( id );
          comboBoxSharedObject.data.dataProvider = dataProvider;
          comboBoxSharedObject.flush();
        }
        dispatchEvent( { type: "userItemAdded", item: text } );
      }
    }
  }
}

private function addSharedObjectItems()
{
  {
    var comboBoxSharedObject = SharedObject.getLocal( id );
    if ( !persistUserList )
    {
      delete comboBoxSharedObject.data.dataProvider;
      return;
    }
    var sharedObjectDataProvider = comboBoxSharedObject.data.dataProvider;

    for ( var p in sharedObjectDataProvider )
    {
      {
        if ( !itemExistsInComboDataProvider( sharedObjectDataProvider[p] ) )
        {
          dataProvider.addItem( sharedObjectDataProvider[p] );
        }
      }
    }
  }
}

private function itemExistsInComboDataProvider( item )
{

```

```

for ( var p in dataProvider )
{
    if ( ( item == dataProvider[p] ) ||
        ( item == dataProvider[p].label ) ||
        ( item.label != undefined && item.label ==
          dataProvider[p].label ) )
        return true;
    }
return false;
}
}
}

```

The only change we made from previous code is in the `itemExistsInComboDataProvider()` method, in which we do some further checks on whether the item being added already exists. These checks are to cover some of the data provider formats that can be used to populate the component, but are not exhaustive. For example, if the client of our component uses the `labelField` or `labelFunction` properties, or a custom renderer, to display the items in the combo box, our component would not accurately track the items.

## Handling the Event

Now that our component broadcasts the event when the user adds an item to the drop-down portion of the control, and we've informed the compiler via the Metadata that the event can be broadcast, we can now change our application to handle the event. We do so by attaching a `userItemAdded` handler, as shown here:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
                xmlns:i2="com.iterationtwo.components" >

    <i2:EditableComboBox id="theEditableCombo"
        editable="true"
        persistUserList="true"
        addUserItemsToList="true"
        userItemAdded="status.text = event.item + ' was
                        added to the combo box item list';" >

        <i2:dataProvider>
            <mx:Array>
                <mx:String>J2EE Developer</mx:String>
                <mx:String>.NET Developer</mx:String>
                <mx:String>Flash Designer</mx:String>
                <mx:String>Graphic Designer</mx:String>
            </mx:Array>
        </i2:dataProvider>
    </i2:EditableComboBox>

    <mx:Label id="status" />

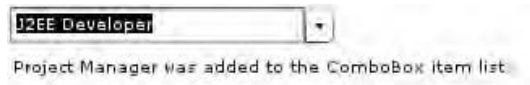
</mx:Application>

```

When the user adds a new entry to the `ComboBox` by pressing `Enter`, our `userItemAdded` handler is called, which sets the text property on our status `Label` by fetching the `item` property we defined on the object broadcast with the event.

Figure 10.2 shows how the application looks to the user after adding a new entry to the component.

**Figure 10.2**  
The editable  
ComboBox custom  
component with event  
handler for user  
additions to the drop-  
down.



## Creating a Custom Container Component

The custom components we created previously are extensions to Flex controls. We can also create our own custom containers, again either by extending an existing one or by creating one from scratch, perhaps by extending the `mx.containers.Container` class.

In Chapter 8, “Building Form-Driven Applications,” we discussed the `<Form>` and `<FormItem>` containers and showed how form fields could be hidden or shown, depending on user entry, such as hiding the Employer fields when the user selects the Unemployed option from the Employment Status ComboBox.

But what if we have many fields onscreen, and want to hide one on a certain user choice and another on a different choice? We could end up with “holes” on the user interface where the field was once shown.

We can get around this quandary by creating a custom container component that extends the Flex `<FormItem>` container, to force the user interface to be laid out again. This can be achieved using the `height` property of the `<FormItem>` container; by setting the `height` property of a container to zero, the children in that container are effectively hidden, and the parent container laid out again with controls below that one moved up the user interface.

Our custom component is defined as follows, in a file called `DynamicFormItem.as`:

```
class com.iterationtwo.components.DynamicFormItem extends
    mx.containers.FormItem
{
    public function set visible( visible:Boolean )
    {
        super.visible = visible;

        if ( !visible )
            height = 0;
        else
            height = undefined;
    }
}
```

We override the implicit setter of the `visible` property of the component to set the `height` of the form item to `0` when the component is hidden. When it’s made visible again, the `height` property is set to `undefined`, so that the component’s preferred height is used when the component is laid out again.

## Implicit Setters and Getters

Implicit setters and getters may be new to J2EE developers; they let developer access object properties directly while still maintaining encapsulation of the data. When the developer using our component sets the visibility of the control by setting the `visible` property to `false`, the implicit setter we defined will be executed.

We use this custom container in the name and address form we created in Chapter 8, except this time we'll show a single Post Code input field if the user is from the UK, or a Zip Code and SSN field if the user is from the U.S. or Canada. Our MXML is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:i2="com.iterationtwo.components">

    <mx:Form id="theForm" verticalGap="0">

        <mx:FormHeading label="Personal Details" />

        <i2:DynamicFormItem id="firstNameItem" label="First name"
            required="true" widthFlex="1" >
            <mx:TextInput id="firstName" />
            <mx:Spacer height="3" />
        </i2:DynamicFormItem>

        <i2:DynamicFormItem id="lastNameItem" label="Last name"
            required="true" widthFlex="1" >
            <mx:TextInput id="lastName" />
            <mx:Spacer height="3" />
        </i2:DynamicFormItem>

        <i2:DynamicFormItem id="address1Item" label="Address"
            required="true" widthFlex="1" >
            <mx:TextInput id="addressLine1" />
            <mx:TextInput id="addressLine2" />
            <mx:TextInput id="addressLine3" />
            <mx:Spacer height="3" />
        </i2:DynamicFormItem>

        <i2:DynamicFormItem id="cityItem" label="City"
            required="true" widthFlex="1" >
            <mx:TextInput id="city" />
            <mx:Spacer height="3" />
        </i2:DynamicFormItem>

        <i2:DynamicFormItem id="countryItem" label="Country"
            required="true" widthFlex="1" >
            <mx:ComboBox id="country" >
                <mx:dataProvider>
                    <mx:Array>
                        <mx:String>USA</mx:String>
                        <mx:String>UK</mx:String>
                        <mx:String>Canada</mx:String>
                    </mx:Array>
                </mx:dataProvider>
            </mx:ComboBox>
        </i2:DynamicFormItem>
    </mx:Form>
</mx:Application>
```

```

        <mx:Spacer height="3" />
    </i2:DynamicFormItem>

    <i2:DynamicFormItem id="postcodeItem" label="Post Code"
        required="true" visible="{ country.text == 'UK' }" >
        <mx:TextInput id="postcode" width="150" />
        <mx:Spacer height="3" />
    </i2:DynamicFormItem>

    <i2:DynamicFormItem id="zipCodeItem" label="Zip Code"
        required="true" visible="{ country.text != 'UK' }" >
        <mx:TextInput id="zipcode" width="150" />
        <mx:Spacer height="3" />
    </i2:DynamicFormItem>

    <i2:DynamicFormItem id="ssnCodeItem" label="SSN"
        required="true" visible="{ country.text != 'UK' }" >
        <mx:TextInput id="ssn" width="150" />
        <mx:Spacer height="3" />
    </i2:DynamicFormItem>

    <i2:DynamicFormItem id="dateOfBirthItem"
        label="Date of Birth" required="true" >
        <mx:DateField id="dateOfBirth" width="150" />
        <mx:Spacer height="3" />
    </i2:DynamicFormItem>

    <i2:DynamicFormItem>
        <mx:Button label="Submit Application" />
        <mx:Spacer height="3" />
    </i2:DynamicFormItem>

</mx:Form>

</mx:Application>

```

### verticalGap and Spacer Control

You may have noticed that the Form using our DynamicFormItem component has its verticalGap property set to 0 and that each DynamicFormItem has a Spacer control with a height of 3—this is to ensure equal separation of the form items.

If we omit the verticalGap property from the Form control and also omit the Spacer controls, when the form is displayed to the user there will be an extra space between the controls where the “hidden” control would be. This is because, although we set the height of the form item to 0, the parent container still knows a component is there, so leaves a space for it as defined by its verticalGap property. By setting this property to 0 and adding our own gap using the Spacer control, we can ensure equal spacing between all the form items, no matter which items are being displayed.

In the three highlighted <DynamicFormItem> instances, we added a binding between the visible property and the value of the country drop-down. When the user changes the country, the visible property is set, which executes the implicit setter on our custom container. That causes the form item height to be set to 0, or undefined, depending on its visibility, which in turn causes the application to be laid out again.

**Figure 10.3** shows the application when the user has selected UK as the country, and **Figure 10.4** is what he sees when making any other choice.

**Figure 10.3**

The name and address form using the dynamic form item, when the user has selected UK from the Country ComboBox.

The screenshot shows a form titled "Personal Details" with the following fields: First name, Last name, Address (three stacked lines), City, Country (dropdown menu showing "UK"), Post Code, and Date of Birth (calendar icon). A "Submit Application" button is at the bottom.

**Figure 10.4**

The name and address form using the dynamic form item, when the user has selected USA or Canada from the Country ComboBox.

The screenshot shows a form titled "Personal Details" with the following fields: First name, Last name, Address (three stacked lines), City, Country (dropdown menu showing "USA"), Zip Code, SSN, and Date of Birth (calendar icon). A "Submit Application" button is at the bottom.

## Creating New Components

The components we created so far have been extensions to those provided with Flex. You can also create your own components from scratch, either in ActionScript only or by using Flash MX 2004 to create a custom component that can then be used with Flex. We'll concentrate on the former; the latter requires knowledge of the MX 2004 product suite and is beyond the scope of this book.

### Creating Components in MX 2004

The documentation that ships with Flex contains a wealth of information detailing how you can build components in MX 2004 for use with Flex.

## Context Menu Components

A context menu is a pop-up menu that appears over a specific part of the application, showing actions that the user can perform on that item in the application. With Microsoft Windows, the context menu is invoked by the user via the right mouse button on the mouse. On a Mac, it's invoked via Control-click. Context menus are added to applications to allow advanced users to reach certain options faster; they should be used only to provide another way for the user to achieve something also available as an option elsewhere in the application, such as a menu bar entry.

To show how to create a component from scratch, we'll create a component that wraps the context menu functionality provided in Flash Player 7. This component will let developers add a default context menu to the application as a whole and add separate context menus to individual components within that application. Each individual item in the menu can be defined as being enabled or disabled and visible or hidden. We can also specify whether a separator line should appear before each item and then attach additional properties to the item—these properties will be part of the event object broadcast when the user selects that menu item.

### Component Usage

Before we create the context menu component, let's consider how it will be used by developers who want to add a context menu to their Flex components. An example is shown here:

```
<i2:ContextMenu id="theButtonContextMenu"
                contextOver="theDataGrid"
                menuItemSelected="handleMenuItemPressed( event );" >
  <i2:dataProvider>
    <mx:Array>
      <mx:Object caption="Add item" enabled="true" visible="true"
                data="addItem" />
      <mx:Object caption="Edit item" enabled="true" visible="true"
                data="editItem" />
      <mx:Object caption="Delete item" enabled="false" visible="true"
                separatorBefore="true" data="deleteItem" />
      <mx:Object caption="Hidden item" enabled="true" visible="false"
                separatorBefore="false" />
      <mx:String>Quit</mx:String>
    </mx:Array>
  </i2:dataProvider>
</i2:ContextMenu>
```

#### Flash Context Menu Built-in Items

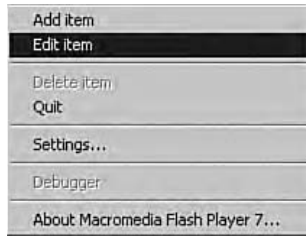
The Flash context menu has a predefined set of built-in items. These can be removed at runtime when we create a context menu, but two items cannot be removed: the Settings item, and the About Macromedia Flash Player 7 item. If you're using the Debug player, as would normally be the case during development, that also remains on the context menu.

The preceding example attaches a context menu to a component with the `id` of `theDataGrid`.

**Figure 10.5** shows how the context menu looks when shown to the user.

**Figure 10.5**

The context menu custom component in use.



#### dataProvider **Namespace**

Notice that the namespace for the `dataProvider` property given to the context menu component is `i2` and not `mx`. This is because `dataProvider` is a property of our component, not a built-in Flex tag. Flex requires that the namespace of a property tag match the namespace of its component tag.

If we consider how a context menu works, we can see the purpose of each of the preceding properties:

- The `contextOver` property is optional. If specified, it defines the component over which this context menu should show. If the developer doesn't provide the `contextOver` property, the context menu will be defined to the application and will show over any component that doesn't have its own context menu. In this example, our context menu will show over a `DataGrid` control.
- The `menuItemSelected` property is a handler we attached to the component. This `menuItemSelected` event is broadcast when the user selects an item from the context menu; it has an event object containing the following properties:
  - The `type` property is always set to "menuItemSelected".
  - The `target` property contains a reference to the context menu component broadcasting the event.
  - The `contextOver` property contains a reference to the control over which the context menu was opened.
  - The `menuItem` property contains a reference to the menu item that was selected.
- The `menuItem` property of the event contains all the properties of the items defined in the `dataProvider` of the component.
- The `dataProvider` property of the component takes an array of objects or strings. When an item in the array is an `<mx:Object>`, the component looks for four special properties:
  - The `caption` property is the text that appears on the menu item. If this property is omitted, the menu item doesn't appear.

- The `enabled` property defines whether the menu item is enabled, allowing the user to select it. The default value is `true`.
- The `visible` property defines whether the menu item is visible. The default value is `true`.
- The `separatorBefore` property defines whether a separator bar should appear before this menu item. The default value is `false`.

The developer using our custom component can provide additional properties in the `<mx:Object>` items in the `dataProvider` array, which are included in the `menuItem` property broadcast with the `menuItemSelected` event. In the preceding example, we added a `data` property to each item, which is available in the `event.menuItem.data` property in our event handler.

If an `<mx:String>` item is part of the `dataProvider` array, it's added to the context menu as an enabled, visible menu item with no separator and no additional data.

Now that you know how a developer can use our context menu component, let's build it.

### Class Definition

We'll create our component as an `ActionScript` class. Here's our class definition, which is stored in a file called `ContextMenu.as` in a `com/iterationtwo/components` directory structure:

```
class com.iterationtwo.components.ContextMenu extends mx.core.UIObject
{
    ...
}
```

We've extended the `mx.core.UIObject` class, so we can use the event-dispatching mechanism of that class. We use the event dispatching to notify clients of the class when the user has selected an item from the context menu.

#### Which Base Class?

When creating custom components, we can extend an existing Flex component, or a more abstract class such as `mx.core.view`, `mx.core.UIComponent`, or `mx.core.UIObject`. In this example, we chose `UIObject` because we have no real visual part of the control; what you choose will depend on what features you want to inherit from the Flex classes.

### Instance Properties

Our context component has two instance properties:

```
public var dataProvider:Object;
public var contextOver:String;
```

The `dataProvider` property accepts the menu items to be shown in the context menu. The `contextOver` property is an optional property used to specify over which component the context menu should show.

## Constructors

The Flex component framework calls the constructor of our component when it's defined within MXML. In the constructor, we define the following:

```
public function ContextMenu()
{
    contextOver = String( mx.core.Application.application );
    parent.addEventListener( "initialize", setUpContextMenu );
}
```

The `mx.core.Application` class has a static class property name `application` that contains a reference to the Flex root application object, which can be referenced from anywhere within your Flex application.

The parent property of a component contains a reference to its parent container as defined in the MXML file or coded in an `ActionScript` class.

In our constructor, we first initialize the `contextOver` property to the name of the application. In `ActionScript`, casting a component to a `String` returns the name of the fully qualified path to that object. The developer using our component can override this property in the MXML file.

The second line requires some further explanation. Remember that our context menu component can be shown over another component, which is defined via the `contextOver` property. For our component to be able to define the context menu on that component, it must have been instantiated. However, when our constructor is called, we don't know whether that object has been instantiated. We do know that when Flex constructs components, it ensures that all its children are initialized before it broadcasts its own `initialize` event; we can use this knowledge to delay the context menu construction to such a point that we can guarantee that the component referenced in `contextOver` has been instantiated.

So we attached a listener to the `initialize` event of our parent control; that event will be broadcast only when both our custom component and any component reference in `contextOver` have been initialized. At that point, it's safe to attach the context menu via the `setUpContextMenu` method of our class.

However, the constructor we defined previously doesn't tell the full story. Flex has some scoping issues that may seem peculiar to Java and C# developers, with particular reference to event handlers and listeners. (In reality, it's not Flex that introduces the scoping issues surrounding event handlers and listeners; it's Flash Player.)

Inside a listener component defined to handle an event, the `this` keyword doesn't contain a reference to the component's class instance, but to the object to which the handler was attached. With the preceding constructor code, `this` in the `setUpContextMenu` method actually contains a reference to the parent property of the class. This is obviously not what we want, particularly if we want to reference `this` in that function or call another method that uses `this`. Thankfully, Flex provides a solution to this problem, encapsulated inside the `mx.utils.Delegate` class.

Here's how our constructor looks now:

```
public function ContextMenu()
{
    contextOver = String( mx.core.Application.application );
    parent.addEventListener( "initialize",
                            mx.utils.Delegate.create( this, setUpContextMenu ) );
}
```

The static `create()` method of `mx.utils.Delegate` returns a function that can be used in place of any listener, and takes two parameters. The second parameter is the method to be called, which is the same as before: `setUpContextMenu`. The first parameter is the scope in which the method should be called, which we pass as `this`, which is the component's class instance in the constructor (the majority of the time, your scope will be `this`).

The result of using the `Delegate` class instance as our listener in our constructor is that when the component's parent broadcasts the `initialize` event, the `setUpContextMenu()` method will component be called; within that method, `this` will refer to our component class instance.

The `Delegate` class is covered in more detail in Chapter 16, "Managing Application Workflow with Events."

## Context Menu Implementation

Let's look at the component implementation of our context menu component. We won't look in detail at the actual `ContextMenu` and `ContextMenuItems` classes used; these are documented in the Macromedia Flash documentation.

The method definition is as follows:

```
private function setUpContextMenu()
{
    ...
}
```

### Flash Player Classes

Some classes, such as the `ContextMenu` class, are built into Flash Player rather than shipped with Flex as ActionScript classes. These classes have stub implementations only (known as *intrinsic*s) in the Flex installation.

Within this method, we first create a new instance of a `ContextMenu` class—Flash Player class that's used to create a context menu. This class is built into Flash Player and doesn't exist in a package structure. Because our custom component is also called `ContextMenu`, this has resulted in a name clash, even if our component resides in its own package. The problem comes to light when we try to create an instance of Flash Player's `ContextMenu` object within our component, as follows:

```
var cm:ContextMenu = new ContextMenu();
```

At first sight, that seems fine. However, the Flex compiler cannot know which `ContextMenu` we're trying to construct—our custom component, which can be referenced without the fully qualified



```

        for ( var q in dataProvider[p] )
            cmi[q] = dataProvider[p][q];
    }

    cm.customItems.push( cmi );
}

cm.customItems.reverse();

```

### Why reverse()?

You may have noticed that we reverse the order of the menu items after we've populated the array from the data provider. This is done because `for ( var p in dataProvider )` returns the items in the reverse order that they were specified in the MXML file. The `reverse()` method of the Array class puts them back in the order they were provided to us.

In the preceding code, we again use the `mx.utils.Delegate` class for our listener to the `ContextMenu` (another Flash Player class) constructor. That listener means that the `contextItemPressed()` method of our class is called when the user selects a menu item from the context menu, ensuring that the scope of `this` is correct within that method. We also copy all properties over from the original data provider object to the context menu item object, so that they're included in the event object we broadcast when the user selects an item.

Finally, within the `setUpContextMenu()` method, we add the context menu we've created to the component it's to show over, whether that's the application or a specific component:

```
mx.core.Application.application[ contextOver ].menu = cm;
```

The `contextItemPressed()` method, which is called when the user selects a menu item from the context menu, is defined as follows:

```

public function contextItemPressed( event, menuObject )
{
    dispatchEvent( { type: "menuItemSelected",
                    contextOver: contextOver,
                    menuItem: menuObject } );
}

```

We broadcast the `menuItemSelected` event, with the properties we previously specified: `type`, `target`, `contextOver`, and `menuItem`.

The final step is to inform the Flex compiler that our component broadcasts that event, so we add the now-familiar line of metadata to the class definition:

```
[Event("menuItemSelected")]
```

That line allows developers to attach a `menuItemSelected` handler to our component.

## The Completed Component

Our context menu custom component is now complete. Here's the full listing:

```
[Event("menuItemSelected")]
```

```

class com.iterationtwo.components.ContextMenu extends mx.core.UIObject
{
    public var dataProvider:Object;
    public var contextOver:String;

    public function ContextMenu()
    {
        contextOver = String( mx.core.Application.application );
        parent.addEventListener( "initialize",
            mx.utils.Delegate.createDelegate( this, setUpContextMenu ) );
    }

    private function setUpContextMenu()
    {
        var cm = new _global["ContextMenu"]();
        cm.hideBuiltInItems();
        var cmi:ContextMenuItems;

        for ( var p in dataProvider )
        {
            var contextMenuItemPressedDelegate =
                mx.utils.Delegate.createDelegate( this, contextItemPressed );

            if ( typeof dataProvider[p] == "string" )
            {
                cmi = new ContextMenuItem( dataProvider[p],
                    contextMenuItemPressedDelegate );
            }
            else
            {
                cmi = new ContextMenuItem( dataProvider[p].caption,
                    contextMenuItemPressedDelegate,
                    dataProvider[p].separatorBefore,
                    dataProvider[p].enabled,
                    dataProvider[p].visible );

                for ( var q in dataProvider[p] )
                    cmi[q] = dataProvider[p][q];
            }

            cm.customItems.push( cmi );
        }

        cm.customItems.reverse();

        mx.core.Application.application[ contextOver ].menu = cm;
    }

    public function contextItemPressed( event, menuObject )
    {
        dispatchEvent( { type: "menuItemSelected",
            contextOver: contextOver,
            menuItem: menuObject } );
    }
}

```

## Using the Context Menu Component

Earlier in this chapter, we showed a small excerpt of code showing how our custom context menu component could be used. Let's look at a fuller example, showing two context menus within the one application: one for the application and one for a component.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:i2="com.iterationtwo.components" >

    <mx:Script>
        var theDataGridProvider = [
            { date:'01-12-2003', description:'Opening Balance',
              debit:'', credit:'', balance:'5000' },
            { date:'03-12-2003', description:'Salary',
              debit:'', credit:'1000', balance:'6000' },
            { date:'05-12-2003', description:'Car Loan',
              debit:'500', credit:'', balance:'5500' }
        ];
    </mx:Script>

    <mx:Model id="theApplicationContextMenuModel">
        <items>
            <item1>Open...</item1>
            <item2>Save...</item2>
        </items>
    </mx:Model>

    <i2:ContextMenu id="theDocContextMenu"
        menuItemSelected="status.text = 'The document menu item
            with caption ' + event.menuItem.caption + ' was
            selected';" >
        <i2:dataProvider>
            { theApplicationContextMenuModel.items }
        </i2:dataProvider>
    </i2:ContextMenu>

    <i2:ContextMenu id="theGridContextMenu" contextOver="theDataGrid"
        menuItemSelected="status.text = 'The grid menu item with
            caption ' + event.menuItem.caption + ' and data ' +
            event.menuItem.data + ' was selected for ' +
            theDataGrid.selectedItem.description;" >
        <i2:dataProvider>
            <mx:Array>
                <mx:Object caption="View details" data="viewDetails" />
                <mx:Object caption="Edit item" data="editItem" />
                <mx:Object caption="Delete item" separatorBefore="true"
                    enabled="false" data="deleteItem" />
            </mx:Array>
        </i2:dataProvider>
    </i2:ContextMenu>

    <mx:Label id="status" widthFlex="1" />

    <mx:DataGrid id="theDataGrid" widthFlex="1" heightFlex="1"
        selectedIndex="0" >
        <mx:dataProvider>
```

```

        { theDataGridProvider }
    </mx:dataProvider>
</mx:DataGrid>

```

```

</mx:Application>

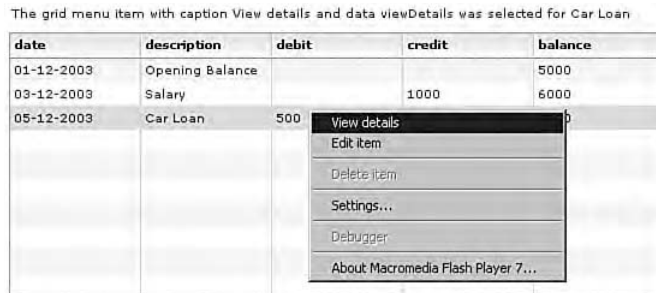
```

This application has two context menus, one that pops up over the DataGrid control, and one that pops up anywhere else in the container. When an item is selected from either menu, the status label control on the application is updated to show the details of the selected item.

**Figure 10.6** shows the context menu over the DataGrid control, having previously selected the item.

**Figure 10.6**

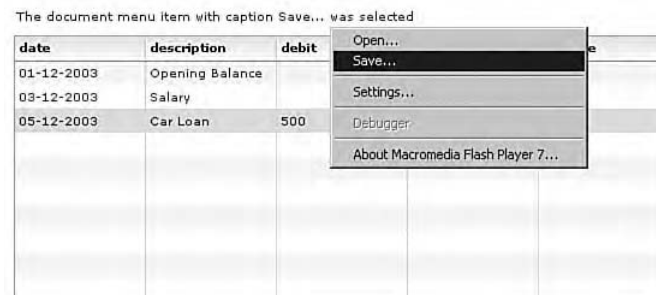
The context menu custom component attached to a DataGrid control.



**Figure 10.7** shows the context menu when the mouse is *not* over the DataGrid. This shows the context menu that was defined without a contextOver property. Again, the status line of the application shows the text when we previously launched the same context menu.

**Figure 10.7**

The context menu custom component attached to the application object.



## Context Menus and Nested Components

Context menus work with Flash Player 7 and above only, but there is still one fairly major restriction to their use. Flash Player cannot show a context menu on a Flex component that resides within a container other than the Application container. This is a result of how Flex compiles MXML into nested movie clips. Unfortunately, until Flash Player supports context menus on nested movie clips, we're stuck with adding them onto the root application object and to components directly within that object.

### Building Flex Components with Flash MX 2004

As well as building components purely in MXML or ActionScript, you can use Flash MX2004 to develop Flex components. The Flex documentation covers the process of doing so in great detail, and requires knowledge of Flash MX 2004, so we won't cover that topic in this book.

We anticipate that a traditional Flash developer is likely to fill this role, passing custom components to a Flex development team. The Flex developer and the Flash developer needn't necessarily be the same person.

## Defining Effects on Custom Components

In Chapter 13, "Behaviors and Effects," you learn how you can apply effects to Flex components, including fading, zooming, or resizing components, when a certain trigger happens, such as the component becoming visible.

We can easily add the ability for developers using our custom components to apply effects to instances of that component upon the broadcasting of our custom events.

Remember our custom editable ComboBox; we broadcast the `userItemAdded` event when the user added a new entry into the drop-down portion of the ComboBox. Metadata on the class definition told the Flex compiler that the `userItemAdded` event was broadcast by our component, as repeated here:

```
[Event("userItemAdded")]
```

Similarly, simply by adding an Effect metadata to our component, we can tell the Flex compiler that developers using our component can add an effect when that event is broadcast. Here is the Effect metadata required:

```
[Effect("userItemAddedEffect")]
```

The name of the Effect metadata must be the same as the event name used to trigger the effect, plus the word `Effect`. This tells the Flex compiler to trigger the effect when the event is broadcast. So, the preceding line of metadata tells the Flex compiler to trigger the `userItemAddedEffect` effect when the `userItemAdded` event is broadcast by our component.

The developer using our component handles that trigger in the same way as all Flex triggers are handled, via a handler attached to the effect in the MXML file. Our MXML file now contains this:

```
<i2:EditableComboBox id="testEditableCombo" width="200" editable="true"
    addUserItemsToList="true" persistUserList="true"
    userItemAddedEffect="WipeRight"
    userItemAdded="status.text = event.item + '
    added to the combo box item list';" >
    ...
</i2:EditableComboBox>
```

Now when the `userItemAdded` event is broadcast, as well as the event handler being invoked, the effect trigger also kicks into action and runs the `WipeRight` effect. That effect is shipped with Flex; you can, of course, substitute that effect for any other effect, including any custom effect.

## Summary

This chapter covered the various ways you can create your own custom components with Flex for use in your applications.

We described how custom components can be defined either in MXML or in ActionScript and gave examples of each, giving an indication as to why you might choose one solution over the other.

We showed how your components can be built from scratch, or you can extend one of the many prebuilt components that ship with Flex. You can even extend your custom component, or someone else's, to add new features.

Although we showed mainly custom controls, we discussed how you could also create custom containers and even a custom application, which could be used to integrate a common architectural framework into all your applications.

Finally, we showed how effects could be applied to our custom controls, so that developers using the controls could change the appearance of them as our control broadcasts events to indicate a change in state.

The component architecture of Flex provides a very powerful mechanism with which we can build our own components, and we fully expect that third-party libraries of Flex components will increase the options and development velocity for Rich Internet Application (RIA) developers.



# CHAPTER 11

## Controlling the Application's Look and Feel

### IN THIS CHAPTER

Branding the Application 255

Styles 255

Fonts 264

Themes 269

### Branding the Application

Macromedia Flex allows the rapid construction of rich client user interfaces using the presupplied containers and controls, each of which exhibit a common “Halo” look and feel. This design is likely to be sufficient for many RIA examples, but most application developers may want to exert some control over the overall application look and feel, and enterprises adopting the Rich Internet proposition may want their Rich Internet Applications (RIAs) to more strongly reflect their branding. This chapter covers the key methods of changing application look and feel within a Flex RIA.

We start by exploring styles, which use the Cascading Style Sheet (CSS) specification to offer the quickest and simplest means of changing the look and feel of an application by using styles and stylesheets. Then we discuss font handling within Flex; we'll ensure that you get a clear understanding of the difference between device fonts and embedded fonts, and the implications and methods of font embedding.

Finally, we take you through themes, which provide the greatest flexibility in defining the look of the user interface, in return for the greatest amount of effort on the part of the graphic designer or Flash designer. Using themes, an RIA development team can change everything: the shape of check boxes and buttons; the look and feel of scroll bars and buttons; and so forth. In this manner, a Flex RIA will be indistinguishable from one created completely by hand with Macromedia Flash.

Let's start our discussion with the broadest of these topics: CSS styling of the user interface.

### Styles

The easiest way to change the look of a Flex component is to modify its appearance by using style properties. Every component in the Flex class library exposes its look and feel through properties, which allow control of such elements as the font color, the font weight or font size in a Label, the background color used in a Tree, or the alternating row colors in a DataGrid.