Adobe® Flex™ 2

Migrating Applications to Flex 2

Adobe

Contents

About Flex Documentation

Migrating Applications to Flex 2 provides information on updating applications written for Flex 1.x to Flex 2.

Contents

Using this manual

This manual can help anyone who has developed Flex applications. You should have an understanding of the architecture and details of the Flex 1.x framework.

Adobe recommends that you begin with Chapter 1, "Getting Started," on page 9. This topic provides several steps that are meant to walk you through performing a large majority of the simple tasks in migration your applications.

You should then examine each of the other topics that provides more in-depth information about changes to particular areas of Adobe® Flex™ 2, including events, effects, and data binding.

Finally, for more complex migration issues, you should examine Chapter 13, "Migration Patterns," on page 173. This topic provides longer examples that provide context to certain difficult migration issues.

Accessing the Flex documentation

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

Documentation set

The Flex documentation set includes the following titles:

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

Viewing online documentation

All Flex documentation is available online in Adobe® Acrobat® Portable Document Format (PDF) files from the Adobe website.

Typographical conventions

The following typographical conventions are used in this book:

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

CHAPTER 1

Getting Started

1

This topic describes the first steps of migrating a Macromedia Flex 1.x application to Adobe Flex 2. This topic describes steps that are meant to be accomplished quickly and resolve most of the warnings and errors that you encounter. When you complete the steps in this topic, you should read other topics in this manual for more information about specific migration operations.

Contents

Introduction

You should undertake the migration process in a series of steps. The earlier steps involve simple tasks, such as finding and replacing, or adding access modifiers. This topic describes these steps. The later steps are more involved and require you to read other topics in this manual. For example, when you convert portions of an application that uses binding, you should read Chapter 5, “Binding,” on page 105.

The basic steps are the following:

- "Step 1: Find and replace" on page 10
- "Step 2: Add access modifiers" on page 14
- "Step 3: Add types" on page 15
- "Step 4: Update events" on page 16
- "Step 6: Put all ActionScript components in packages" on page 17
- "Step 7: Update data services" on page 18
- "Step 8: Charts" on page 18
- "Step 9: Overrides" on page 18
- "Step 10: Binding" on page 19
- "Step 11: API updates" on page 19

The remaining sections of this topic describe these steps in detail.

Step 1: Find and replace

There are many simple operations that you can do to your application to minimize the number of warnings and errors you get when you first attempt to compile your application. In many cases, these operations require that you do a find and replace.

You should not have to spend much time performing these tasks because the differences between the Flex 1.x and Flex 2 syntax is generally minor but necessary. These operations include the following:

- "Application namespace" on page 11
- "Void" on page 11
- "Newline" on page 11
- "Effects/behaviors" on page 14
- "Application and container initialization" on page 12
- "Alpha and scale properties" on page 12
- "Event.currentTarget" on page 12
- "Uninitialized values" on page 12
- "Replace _root" on page 13
- "Application namespace" on page 11
- "Effects/behaviors" on page 14
- "getURL() method" on page 14

This section describes only the most common targets of finding and replacing. There are many other members of Flex classes that have changed that are not mentioned here. For a complete list, see Chapter 3, “Flex Classes,” on page 41.

Application namespace

Change the MXML namespace. Change the following:

```
xmlns:mx="http://www.macromedia.com/2003/mxml"
```

to this:

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

For example:

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

Void

Replace `Void` (with a capital V) with `void` (with a lower-case v).

Newline

The `newline` constant has been removed. In ActionScript, use “\n” to add a carriage return in your Strings. In an MXML tag, use the `&#13;` XML character entity to add a carriage return.

Color value formats

Replace all occurrences of 0x with # in CSS style sheets or `<mx:Style>` tag blocks. The supported color value formats have changed; for example:

```
.b1 { color: red; }           // Valid
.b2 { color: #FF0000; }       // Valid
.b3 { color: 0xFF0000; }      // Invalid
```

In calls to the `setStyle()` method, you can prefix RRGGBB color values with 0x or #, but you must put quotation marks around constants and # values; for example:

```
b1.setStyle("color",0xFF0000);    // Valid
b2.setStyle("color","red");       // Valid
b3.setStyle("color","#FF0000");   // Valid
b4.setStyle("color",red);         // Invalid
b5.setStyle("color",#FF0000);     // Invalid
```

For more information on changes to the supported color value formats, see “Using colors” on page 121.

Application and container initialization

The `initialize` event is now dispatched later in the startup and component creation life cycle. In particular, it is now dispatched after the object's children have been created. If your event handler assumes that the object's children have already been created, you can use the `initialize` event. If your event handler requires that the object's children have been processed by the LayoutManager class, use the `creationComplete` event; for example:

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

Alpha and scale properties

The `scaleX`, `scaleY`, and `alpha` properties now range from 0.0 to 1.0, rather than 1 to 100. This is also true for the `Fade.alphaFrom`, `Fade.alphaTo`, `Zoom.zoomFrom`, and `Zoom.zoomTo` properties.

Event.currentTarget

Use the `currentTarget` property rather than the `target` property of the Event object. The Event object now has two properties: `target` and `currentTarget`. The former is the object that originally dispatched the event, and the latter is the object to which your event handler is attached. The two are often the same, but they may differ if the event has bubbled up from a child object.

In most cases, you will want to change the `target` property to the `currentTarget` property, because the object that was previously the target is now most likely the current target; for example:

```
switch (event.currentTarget.className) { ... }
```

For more information, see "Using the target property" on page 113.

Uninitialized values

Remove checks against the `undefined` value. In Flex 2, `undefined` is only for use with the * (no-type) "type". Other types can no longer store the `undefined` value. If you assign `undefined` to other types, it will be coerced into `null`, `NaN`, `0`, or `false`. You can use the `isNaN()` method to check if a variable is `NaN`.

If you do not initialize a variable at all, the initial value is different than in Flex 1.x. For more information, see "Initializing variables" on page 29.

Replace _root

Remove the use of "`_root`" from your application code if you used it to access the Application instance. In Flex 1.x, you could use `_root` to refer to the main application from anywhere in your application or one of the components. This was never a recommended practice, but was a convenient way to access the application root. To access the Application instance from anywhere in your application, use `Application.application`.

```
import mx.core.Application;
function myFunction():void {
   //_root.ta1.text = "Thank you!"; // Flex 1.x
   Application.application.ta1.text = "Thank you!"; // Flex 2
}
```

Also, "_global" and "_level0" no longer exist.

Alerts

The `Application.alert()` convenience method was removed. You must now use the `mx.controls.Alert.show()` method. You import the mx.controls.Alert class and call the `Alert.show()` method, as the following example shows:

```
import mx.controls.Alert;
public function myEventHandler(event:Event):void {
   Alert.show("An event occurred!");
}
```

If you call the `show()` method in an MXML tag, then you must also include a script block that imports the mx.controls.Alert class, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
       import mx.controls.Alert;
  ]]></mx:Script>
  <mx:Button id="b1" label="Show Style" click="Alert.show('Alert!');"/>
</mx:Application>
```

Effects/behaviors

For each effect, such as Fade, Sequence, and Parallel, change the `name` property to `id`. Also, remove the `<mx:Effect>` tags; for example:

```
<mx:Sequence id="myWipes">
   <mx:WipeLeft/>
   <mx:WipeRight/>
   <mx:WipeUp/>
   <mx:WipeDown/>
</mx:Sequence>
```

For more information, see Chapter 8, "Behaviors," on page 129.

getURL() method

Replace the `getURL()` method with the `navigateToURL()` method in the flash.net package. This global method takes a URLRequest object; for example:

```
var url:URLRequest = new URLRequest("http://mysite.com");
navigateToURL(url,"_self");
```

For more information, see Chapter 34, "Communicating with the Wrapper," in *Flex 2 Developer's Guide*.

Step 2: Add access modifiers

You must add access modifiers to all your properties, variables, methods, and classes. Available access modifiers are `public`, `internal`, `private`, or `protected`.

When you first write or port an application, it is easiest to set every method and property to `public`. You can then revisit the application when it is working and begin restricting access by adding the `private` identifier where necessary.

The following table shows some common situations where you add modifiers:

Flex 1.x	Flex 2

`function processVariables() {`
`  // Returns a Boolean`
`}`
	`public` `function processVariables():Boolean {`
	`  // Returns a Boolean`
	`}`

`function getBalance() { ... }`	**`private`** `function getBalance():Number { ... }`
`var s = "My name is Fred.";`	**`private`** `var s:String = "My name is Fred.";`
`class MyButton extends Button { ... }`	**`public`** `class MyButton extends Button { ... }`

The default access modifier for methods, variables, and classes is `internal`. This means that all classes in the same package can access them, but classes outside of the package cannot. However, the Flex compiler issues a warning if you do not specify any access modifier.

For more information about access modifiers, see "Access modifiers" on page 27. For information about disabling warnings, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Step 3: Add types

All variables, properties, method arguments, and method return types should now be typed. To find variables and properties, search for the keyword "var" to locate places where you should type variables and properties.

The following table shows common variable typing tasks:

Flex 1.x	Flex 2
`var s = "Title Page";`	`public var s:String = "Title Page";`
`var myType = event.type;`	`public var myType:String = String(event.type);`
`item = event.target.selectedItem;`	`item = ThumbnailView(event.target.selectedItem);`
`public function myHandler(event):void`	`public function myHandler(event:Event):void`

To find methods and method arguments, search for the keyword `function` and add a return type if that method returns a value and type each method argument; for example:

Flex 1.x:

```
function getAnswer(myString) {
   ...
   return myString;
}
```

Flex 2:

```
public function getAnswer(myString:String):String {
   ...
   return myString;
}
```

For more information, see "Explicit typing" on page 32.

Step 4: Update events

The Event model changed in Flex 2. However, you can migrate most event handlers with minimal effort. This section describes the most popular changes that you must make to your event handling code. For more information about migrating events, see Chapter 6, “Events,” on page 111.

Specify types

Events are now more strongly typed than in Flex 1.x. As a result, you should specify the object type in the event listener function. For example, previously you could write the following:

```
function myListener(event) { var s = event.type; }
```

Now, you must specify an event object in the function’s signature; for example:

```
private function myListener(event:Event):void { var s = event.type; }
```

Where possible, make the events as specific as possible. If the previous example was a listener for click events, specify a MouseEvent as the type:

```
private function myListener(event:MouseEvent):void { ... }
```

Dispatch custom events

When you dispatch custom events, you must declare a new Event object rather than use a generic object; for example:

```
click="dispatchEvent(new Event('checkOut'))"
```

Remove delegates

You are no longer required to wrap a listener function with a Delegate to maintain application scope. You can remove the Delegate from the following line:

```
b1.addEventListener("click", mx.utils.Delegate.create(this,myListener));
```

So the line appears as follows:

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

Step 5: Import classes for package-level functions

When you use package-level functions, you must import the package. For example, in Flex 1.x you could call the `show()` method in the following way:

```
mx.controls.Alert.show("This is an Alert!");
```

In Flex 2, you must import the package before calling the function, as the following example shows:

```
import mx.controls.Alert;
Alert.show("This is an Alert!");
```

Step 6: Put all ActionScript components in packages

You must wrap all ActionScript components in a `package` statement, even if the component is located in the same directory as the main application. If the component is in the same directory as the main application, you can use an unnamed package, but the `package` statement must be the first line of the component's file; for example:

```
package {
   public class MyClass {
      // Class definition
   }
} // Close package
```

Adobe recommends that you use unique package names so that there are no duplicate class names in your source paths. The generally accepted syntax is to use the reverse URL technique; for example:

```
com.yourcompany.MyPackage
```

Package names must match the directory hierarchy. For example, if you have the following component used by your main application:

```
/myfiles/MainApp.mxml
/myfiles/mycomponents/TrivialComponent.mxml
```

Your package name must be mycomponents, as the following example shows:

```
package mycomponents {
   ...
}
```

For more information about the `package` statement syntax, see "Package statement syntax" on page 26.

Step 7: Update data services

The RemoteObject, HTTPService, and WebService MXML services are now known as RPC services. The RemoteObject tag is not functional unless you use Adobe Flex Data Services. You must replace use of this with another tag.

You can still use the HTTPService and WebService tags, but you can only access resources on a server that is in the same domain as the Flex application or from a server that has a crossdomain.xml file installed on it. This file must allow access to the requesting application's domain. In addition, you must set `useProxy=false` in the MXML tag (the default).

For more information on using data services, see Chapter 9, "Data Services," on page 135.

Step 8: Charts

If you are using Adobe Flex 2 SDK, you must now install a separate SWC file to use charts. For more information, see the installation instructions in the Adobe Flex Charting' readme.txt file.

If you are using the Flex Data Services server, charts are included.

The way charts are implemented also changed significantly. For more information, see "Charting" on page 165.

Step 9: Overrides

Whenever a method on a subclass overrides a similarly named method on a superclass, the declaration of the subclass's method must now be prefixed by `override`. For more information, see "Overriding a method" on page 163.

Step 10: Binding

To make a user-defined variable bindable, you must now explicitly identify it by adding the `[Bindable]` metadata tag to the property; for example:

```
[Bindable]
public var catalog:Array;
```

In addition, the `<mx:Binding>` tag must be moved to the top level. The following example compiled in Flex 1.5:

```
<mx:HBox>
  <mx:Label id="myLabel"/>
  <mx:Label id="my2ndLabel" text="hello"/>
  <mx:Binding source="my2ndLabel.text" destination="myLabel.text"/>
</mx:HBox>
```

In Flex 2, you must move the tag to the top level, as the following example shows:

```
<mx:Binding source="my2ndLabel.text" destination="myLabel.text"/>
<mx:HBox>
  <mx:Label id="myLabel"/>
  <mx:Label id="my2ndLabel" text="hello"/>
</mx:HBox>
```

For more information, see Chapter 5, "Binding".

Step 11: API updates

The Flex ActionScript API has been updated to be more user-friendly. Changes include enforcing proper capitalization, making class and property names clearer and more intuitive, eliminating redundant classes and properties, and unifying common properties across controls.

The following specific areas changed:

- Event names. All event names and their constants were changed to present tense. For example, the childAdded event is now childAdd, and its constant, CHILD_ADDED, is now CHILD_ADD.
- Camel-casing. Package, method, and property names now consistently use camel-case. For example, the mx.containers.accordionclasses package name is now mx.containers.accordionClasses.
- Renderers. The terms *cell*, *data*, and *row* have been changed to *item* for renderers. For example, cellFocusIn is now itemFocusIn, and the charting event mouseOverData is now itemMouseOver.

- Expanded abbreviations. Abbreviations in property and method names have been expanded where practical. For example, the `hPosition` property is now `horizontalPosition`.
- Property names. Some properties have been renamed so that their function is more evident. For example, the Boolean `multipleSelection` is now `allowMultipleSelection`.
- Chart skins. The chart skins are now referred to as renderers.
- Unnecessary interfaces and classes. Unnecessary interfaces and classes have been removed and their functionality has been moved to other interfaces and classes where necessary.

For a complete list of changes, see Chapter 3, “Flex Classes,” on page 41. This topic presents these changes in tabular format.

CHAPTER 2

ActionScript 2.0 to 3.0

2

The ActionScript language has undergone a complete redesign. It is now a more robust, type-safe, and usable language. This topic provides an overview of changes to the ActionScript language. For information about using ActionScript 3.0, see *Programming ActionScript 3.0*. For a complete reference on using ActionScript 3.0, see the *ActionScript 3.0 Language Reference*.

Contents

Overview

This section lists some of the most common changes that you will encounter when migrating ActionScript 2.0 to ActionScript 3.0. The remaining sections provide more details about specific changes in the language.

- Most classes are sealed (non-dynamic); as a result, you cannot get or set properties, or call methods, that weren't declared.
- Declarations are no longer public by default.
- You cannot get or set a property, or call a method, on an object reference which is null or undefined.
- Method overrides must be marked `override`, and the signature must match exactly.

- A subclass cant have a var with the same name as one visible from the superclass, and you can't override a var.
- You must declare a type for everything or you'll get a compiler warning.
- Accessing methods or properties of the target of an event object won't compile unless you cast event.target to the type of the target. For example, event.target.foo must be changed to MyComponent(event.target).foo.
- You cannot do a for in loop on every object to see the object's properties. This only works on dynamic objects now. An alternative is to use E4X to do object introspection.
- All classes must be in a package or they will only be accessible from the current script.
- You should put return types on all of your functions.
- Array is a final class.
- The `trace()` method is now in the flash.util.trace package.
- The Timer class in flash.util replaces setInterval and setTimeout.

Usability improvements

This section describes changes to the ActionScript language that improve usability. Most changes described here affect the syntax but do not add functionality. This section describes general changes to the language, and not specifics, although it includes some illustrative examples.

The goal of these usability improvements was to provide consistency. This makes the language easier to understand and reduces the amount of time it takes developers to learn it. The less special case rules need to be remembered, the better.

There are some places where a better developer experience was achieved by bending the rules. For example, ActionScript 3.0 puts enumerations into inner classes. Putting all keycode constants into an inner class of Keyboard, however, creates too verbose a language: Keyboard.KeyCode.UP instead of Keyboard.UP. To create a better developer experience, it was necessary to bend the rules slightly.

Some of these changes come at the price of higher migration costs, but the longer-term benefit is a more robust language.

Capitalization of identifiers

The ActionScript 3.0 naming conventions match the Flex application model and the ECMAScript standard. All identifiers are in "camel case." This means that an identifier's characters are in lowercase, except that the first letter of each word in the identifier is capitalized. For example, ExternalInterface.

Class names are fully capitalized using camel case. The first letter of a class name is always a capital letter. Names of nested classes start with a capital letter, just like any other class name.

Members of classes, with the exception of nested classes, start instead with a lowercase letter. For instance, `addChildAt()`.

Package names are not capitalized at all. For instance, flash.display and flash.external.

Acronyms in class name identifiers are fully capitalized, even if the letters of the acronym are adjacent to another capitalized word in the identifier. For example, URLRequest and IMEEvent.

For members of classes, acronyms are fully capitalized, except when the acronym is the first word of the identifier. When this occurs, the acronym is entirely in lowercase. For example, `swfVersion` and `url`. When the acronym is a subsequent word, it is fully capitalized. For example, `loaderURL()` and `navigateToURL()`.

Constants are one place where we do not use camel case. The entire identifier should be in uppercase, with underscores separating each word. For instance, MouseEvent.CLICK. The rule is similar to Java and ECMAScript conventions.

Package reorganization

Some packages have been changed to make the classes within them located in a more intuitive place. In addition, special care was taken to avoid having too many packages in the Flash Player API. If the ActionScript packages were too granular, you would have a hard time finding the classes you want to use. For some examples, see "Global functions" on page 34.

Accessors

Like C#, ECMAScript 4 supports using accessors. Accessors were also a feature of ActionScript since Flash Player 6. Instead of hand-coding the `getX()` and `setX()` methods, developers can declare a property in such a way as to be interpreted as an accessor. Getting or setting the value of this accessor property invokes the associated get or set function.

ActionScript 2.0 had many instances where getter and setter methods existed rather than accessors. Now, ActionScript 3.0 uses accessors wherever possible. Unless a function has arguments, it was converted to an accessor.

Methods that return a Boolean such as `Socket.isConnected()` were converted to accessors, but retained the "is" or "has" prefix. For example, a method called methods would be converted to the `Socket.isConnected` property.

Internal functionality marked private

ActionScript 3.0 includes a greater number of classes and members that are marked private. Prior to this, maybe private members were not marked appropriately and functionality that was intended to be used internally was exposed.

Naming conflicts with Flex classes

The Flex class library and the Flash Player API share similar class names, such as Buttons, Images, and TextFields.

It is important that classes in the Flex classes and the Flash Player API not have the exact same names. Even though the package system gives classes "long names" and helps partition conflicting names away from each other, you might import many packages. If Flex and Flash Player both have a class named Button, and you import both packages, the Flex compiler throws ambiguity errors.

As a general rule, the Flex class library and Flash Player API do not use the same name for a class any more.

Integer constants in enumerations

In ActionScript 2.0, enumerations were often expressed using string constants. For instance, the `TextField.align` property could be set to the strings `left`, `center`, or `right`.

ActionScript 3.0 generally uses integer constants instead of strings. Integer constants are declared as `public static const` members of a class, usually with type uint. The naming should be all uppercase with underscores separating words.

Using integer constants has benefits for performance, and makes it possible for typos and other usage errors to be detected at compile-time.

Abbreviations

ActionScript 3.0 contains fewer abbreviations in method and property names. The names are as descriptive as possible. For example, the `mapPt()` method is now `mapPoint()`.

Some abbreviations are still used in ActionScript 3.0. For example, the `getBounds()` method was not changed to `getBoundaries()` because it is shorter but just as descriptive.

In some cases, the use of abbreviation was preferred. For example, the `getBoundsOfCharacter()` method was changed to `getBoundsOfChar()` because the term char is used in many other places in the API, such as `String.charAt`.

Consistent use of prefixes

Prefixes on member variables in ActionScript 3.0 have been removed. For interfaces, the prefix “I” has been added; for example IEventDispatcher. Class names do not have prefixes.

The prefixes “is” and “has” are used on methods and accessors that test conditions. They ask a question of the framework. For example, `isXMLName()` asks “Is this an XML name?”. As a result, “is” and “has” prefixes are used only for reading properties and not writing.

The prefix “use” may be used to indicate a Boolean property that can be modified. For example, `useHandCursor()`.

Type annotations

All variables and methods now have type annotations, and those type annotations are as precise as possible. The Object type is only used when there is no reasonable alternative, or where attempting to strongly type the value makes the language more difficult to use. For more information on type annotations, see “Typing” on page 32.

The this keyword

The use of the `this` keyword has been made more consistent. It now refers to the instance of the class that the current method is in.

Classes and packages

This section describes changes to the way you write classes, interfaces, and packages in ActionScript.

Using packages

The `package` statement syntax has changed. In addition, you are now required to put custom ActionScript components inside packages.

If you do not put a class inside a package, then the class is only visible in the current script. For example:

```
class Foo {}
```

is a class that is not in any package, so it is only visible in the current script. If you put a class in an unnamed package:

```
package { public class Foo {} }
```

Then you can access the class from any script. All scripts import the unnamed package by default.

Package statement syntax

You now use a package statement rather than dot notation syntax to declare classes inside packages. You add `package` statements before imports, wrapped around the entire class. For example, to place the Button class in the mx.controls package in Flex 1.5:

```
class mx.controls.Button extends mx.core.UIComponent {
   function Button() {
   }
}
```

This implicitly declared that class Button was in package mx.controls. To place the Button class in the mx.controls package in Flex 2:

```
package mx.controls {
   public class Button extends mx.core.UIComponent {
      public function Button() {
      }
   }
}
```

Custom component packages

ActionScript 3.0 now requires that all ActionScript components be inside packages. These packages can be unnamed; for example:

```
package {
   class MyClass { ... }
}
```

Using classes

This section describes changes to ActionScript classes.

Access modifiers

The new `internal` access modifier refers to a different namespace in each package definition, and is not defined outside of a package definition (that is, in global code).

If a class in a package isn't marked `public` or `private`, then it defaults to `internal`. The class cannot be accessed by classes in other packages; this is the same as the `protected` modifier in Java. Accessing internal classes from outside of the package causes a ReferenceError at run time.

If you do not put any namespace (public, private, internal, or user-defined) on a declaration, the Flex compiler throws a warning.

The following table summarizes the access modifiers:

Access Modifier	Description
private	Only accessible to the class.
public	Accessible from anywhere.
protected	Private to classes and subclasses.
internal	Private to other classes in the package.

Inside a package, the default access specifier is `internal`. Outside of a package, the default access specifier is `public`.

Class identifiers

The form ClassIdentifiers:ClassIdentifiers.Identifier has been deprecated and results in a compile-time warning. It is equivalent to declaring the class name Identifier in the package ClassIdentifiers. For example:

```
class P.A {}      // ActionScript 2.0
package P {       // ActionScript 3.0
    class A {}
}
```

super()

You can only use a `super()` statement inside a constructor. It is a syntax error to use a `super()` statement anywhere else in a program. Previous versions of ActionScript allowed `super()` statements to be used anywhere in a class (except in a static method).

Using external files

This section describes changes to embedding, including, and importing external resources with ActionScript in your Flex applications.

include

In ActionScript 2.0, the `include` keyword was preceded by an octothorp: `#include`. You now use the keyword without the octothorp, and end the line with a semi-colon. For example:

ActionScript 2.0:

```
#include "../scripts/thescript.as"
```

ActionScript 3.0:

```
include "../scripts/thescript.as";
```

import

The `import` keyword lets you reference classes from other packages in your application. For example, to use a `trace()` statement in your custom ActionScript class, you import the flash.util.trace class. The `import` syntax is as follows:

```
import class;
```

For example:

```
import flash.util.trace;
```

You can optionally import entire packages using the wildcard syntax, as the following example shows:

```
import flash.util.*;
```

However, best practices dictate that you only import the classes you need and not the entire package. Doing so is better for performance and debugging.

ActionScript 3.0 uses a number of implicit imports to provide direct access to common global functions such as `trace()`. In ActionScript 3.0, the number of implicit imports has been reduced. For more information, see "Explicit imports" on page 34.

It is important to understand that you should import classes with an import statement rather than use the full classname in your code. For example, do this:

```
import mx.formatters.*;
public var f:NumberFormatter = new NumberFormatter();
```

Rather than this:

```
public var f:mx.formatters.NumberFormatter = new
  mx.formatters.NumberFormatter();
```

Doing the latter results in a “Type annotation is not a compile-time constant” error.

Embed

The Embed syntax in ActionScript is `[Embed(params)]`, and this metadata must be used over a variable. The variable should be of type Class; for example:

```
[Embed(source="holdon.mp3")]
var sndCls:Class;
public function playSound():void {
   var snd:flash.media.Sound = new sndCls();
   snd.play();
}
```

This works in a very similar way to what was implemented in Flex 1.x. The main difference is that instead of typing the embedded variable String, you type it as Class. In addition, instantiation relies on the `new` operator rather than the `attachMovie()` and `attachSound()` methods.

In Flex 1.x, the `[Embed]` metadata was placed before a var, and the var received the linkage ID that was associated with the asset. In Flex 2, you put the `[Embed]` metadata before a class definition to associate the class with an asset. This change makes the linkage ID unnecessary; instead, the asset is manifested to the object model as a class.

For backwards compatibility, the variable associated with an `[Embed]` can be of type String. However, this requires you to get the class via the `getDefinitionByName()` method. For example:

```
[Embed(source="holdon.mp3")]
var sndStr:String;
public function playSound():void {
   var sndClass:Class = getDefinitionByName(sndStr);
   var snd:flash.media.Sound = new sndCls();
   snd.play();
}
```

Initializing variables

The default values of uninitialized typed variables (and arguments and properties) was always `undefined` in ActionScript 2.0, and is almost never `undefined` in ActionScript 3.0. In ActionScript 3.0, `undefined` is only for use with type Object. Other types can no longer store the `undefined` value. If you assign `undefined` to other types, it will be coerced into `null`, `NaN`, `0`, or `false`.

Default values

The following example shows the default values for variables of different types:

```
var dog:int;    // defaults to 0
var dog:uint;   // defaults to 0
var dog:Boolean;// defaults to false
var dog:Number; // defaults to NaN
var dog:Object; // defaults to undefined
var cat;        // type defaults to Object, so default value is undefined
```

For all other classes (such as String, XML, MovieClip, Sprite, or any user-defined class), `null` is the default value for uninitialized variables of that type.

```
var dog:String;           // defaults to null
var dog:UserDefinedClass; // defaults to null
var dog:MovieClip;        // defaults to null
```

With the exception of Object, the types with a non-null default value are the types which can never hold `null` as a value. They get what a typecast of `null` to that type would result in.

To check for a possibly uninitialized variable, compare against the defaults described in "Default values" on page 30. Alternatively, assign an initial, impossible value to your variable's declaration which you can then test against; for example:

```
var length:int = -1;
  if (someTest())
  length = 2;
if (someOtherTest())
  length = 3;
if (length == -1)
  length = lastResort();
```

For instances of classes, use the following pattern:

```
if (o)          // ActionScript 3.0
if (o != null)  // ActionScript 2.0
```

and

```
if (!o)         // ActionScript 3.0
if (o == null)  // ActionScript 2.0
```

Because a Boolean can only be true or false, and never undefined, write the following when checking if a Boolean is true:

```
if (b)          // ActionScript 3.0
if (b == true)  // ActionScript 2.0
if (b != false) // ActionScript 2.0
```

About undefined

In general, ActionScript 2.0 allowed accessing undeclared variables whose value had not yet been set. By default, their value was `undefined`. In ActionScript 3.0, accessing undeclared variables results in a ReferenceError. You can use the `hasOwnProperty()` method to check if a variable has been declared:

```
if (hasOwnProperty('b') == false)
  b = 20;
if (someObj.hasOwnProperty('myDynamicProp') == false)
  someObj.myDynamicProp = 22;
```

Alternately, you can use a try/catch block to catch the ReferenceError; for example:

```
try {
  bb;
} catch(x:ReferenceError) {
  print("no bb");
  bb = 20;
}
```

Another alternative is to check if the `typeof` the variable is `undefined`.

```
if (typeof bb == undefined)
  bb = 22;
```

The `typeof` keyword does not throw a ReferenceError in this case, but rather returns `undefined`.

For a dynamic class such as Object, the compiler expects that instances will often have dynamic properties added to them. The compiler does not throw a ReferenceError when accessing an undeclared property of a variable if its value is an instance of a dynamic class; for example:

```
var a:Object = new Object();
if (a.foo == undefined)      // Does not throw a ReferenceError.
  a.foo = 22;
```

About NaN

`NaN` (Not a Number) is a special instance of Number used to indicate a value which is outside the range of valid Numbers. Any Boolean valued comparison involving `NaN` (such as ==, ===, =>, and >) always returns `false` because any Number operation involving `NaN` is bogus. This is exactly how C++, Java, and other languages treat `NaN` as well.

For comparisons with `NaN`, use the global `isNaN()` method:

```
var a:Number;
if (isNaN(a))
  a = 22;
```

If you previously checked against undefined for a Number, you use a similar syntax; for example:

```
if (n == undefined) // ActionScript 2.0
if (isNaN(n))       // ActionScript 3.0
```

Typing

ActionScript 3.0 is more strongly typed than previous versions of ActionScript. This section describes changes to the rules of typing.

Explicit typing

Flex 2 checks for type correctness for class and package property/methods at compile-time, and enforces types at run time. By using stricter typing, you increase performance of your application and strengthen the compile-time error checking.

If a variable is untyped, it will be treated as if it is type Object, which is slower than a Number, int, uint, String, or Boolean. You should use Object only when absolutely necessary, and you should never leave a variable untyped.

Use the most restrictive type that will work. Do not use Number if you can use int, and do not use int if you can use uint. Use Class if something is a class reference.

In many cases, you will need to cast a general type to something more specific to get the benefits of strong typing. Omitting types causes the Flex compiler to throw a warning, so your applications will compile without this step.

The compiler enforces type checking wherever possible, but due to the dynamic nature of the language, it does not checking style variable access or variables and functions declared outside of classes (because they could be redefined dynamically at run time).

In ActionScript 3.0, type checking is performed at run time. In previous versions of ActionScript, it was only checked at compile time. The following example shows how the compiler will react

```
class a {
  var b:String;
}
var c = new a();
c.b = 22;     // Results in a compiler error.
c["b"] = 22;  // ActionScript 2.0 would allow this, but ActionScript 3.0
              // throws a TypeError exception.
```

The arguments assigned to method parameters must have compatible number and types.

Type detection

To perform type detection, you should use `is` rather than `typeof` or `instanceof`. The `is` function examines types, whereas the `instanceof` function looks at the prototype chain. The `instanceof` function has been deprecated.

For example:

```
class B {
   public function B() {}
}

class A extends B {
   public function A() {
     super();
   }
}

class Main extends MovieClip {
   public function Main() {
     var a:A = new A();
     trace(a instanceof B); // false; instanceof is deprecated.
     trace(a instanceof A); // false; instanceof is deprecated.
     trace(a is B); // true
     trace(a is A); // true
   }
}
```

Primitive types

In ActionScript 3.0, there is no longer a distinction between primitive types such as strings and instances of the String class. All strings are instances of String; every datatype is an instance of some class. The same is true for numbers and Booleans. For example:

```
print("test" == String("test"));       // true
print("test" === String("test"));      // true
print(45 == new Number(45));           // true
print(45 === new Number(45));          // true
print("one;two;three".substring(4,7));// "two"
print((255).toString(16));             // "ff"
```

Non-assignment expressions

A non-assignment expression is any expression where a value is not assigned to a property (or a variable or argument); for example:

```
var myVar : hintString ? String : Number;
```

`(hintString ? String:Number)` is the non-assignment expression. The variable `myVar` is dynamically typed to be String if `hintString` is `true`, else its type will be Number. You can use a non-assignment expression anywhere that you would use a type-assignment expression.

Global functions

Most global functions have changed packages. In addition, they are no longer implicitly imported. This section describes some changes to the global functions.

Explicit imports

In most cases, you must import the classes to use global functions. For example, to use functions in the System.* package, you now import flash.system.*. The global functions in System.* were previously accessible in your `<mx:Script>` blocks without explicit imports.

The following table lists common functions that you must now explicitly add an `import` statement to use:

Function	Package to import
ContextMenu	import flash.ui.ContextMenu;
EventDispatcher	import flash.events.EventDispatcher; (was mx.events.EventDispatcher mix-in)
LocalConnection	import flash.net.LocalConnection;
MovieClip	import flash.display.MovieClip;
PopUpManager	import mx.managers.PopUpManager;
XML	import flash.xml.XMLNode;

For more information on what packages global functions are in, see the global function's entry in *ActionScript 3.0 Language Reference*.

The `trace()` function is an exception. It is in the flash.util package, but is *implicitly* imported.

Function changes

Some global functions have been removed and replaced with new functions. The following table shows removed global functions and their ActionScript 3.0 equivalents:

ActionScript 2.0	ActionScript 3.0
`chr(num)`	`String.fromCharCode(num)`
`int(expr)`	`Math.round(expr)`
`length(expr)`	`expr.length`
`mbchr(num)`	`String.fromCharCode(num)`
`mblength(string)`	`string.length`
`mbord(char)`	`String(char).charCodeAt(0)`
`mbsubstring(string, index, count)`	`string.substr(index,count)`
`ord`	`String(char).charCodeAt(0)`
`random`	`Math.random`
`subString(string, index, count)`	`string.substr(index,count)`
`getProperty(target, propertyName)`	`target.propertyName`
`setProperty(target,propertyName,value)`	`target.propertyName = value`
`object.addProperty( prop:String, getFunc:Function, setFunc:Function)`	Class definition should use: `function get prop() { ... } function set prop() { ... }`

Deprecated features

The following features are deprecated:

- The `Object.registerClass()` method no longer works on ActionScript 1.0 prototype objects, only on ActionScript 2.0 classes.
- Prototype inheritance no longer works for the built-in objects.
- Features such as __resolve no longer works the same way.
- ASNative is no longer supported.
- #initclip no longer supported.
- _global may not be supported (use static variables of classes).
- Most classes are sealed, not dynamic, and cannot have properties added to them.
- Top-level functions such as `gotoAndPlay()` are not supported or changed.

Miscellaneous

This section describes miscellaneous changes to the ActionScript language.

MovieClip

MovieClip has been replaced by Sprite as the parent class of the base object for Flex controls, UIComponent.

The following table lists changes to methods of MovieClip:

ActionScript 2.0 method	ActionScript 3.0 equivalent
`beginMeshFill()`	The `beginMeshFill()` method has been removed. There is no ActionScript 3.0 equivalent.
`swapDepths()`	Target paths are not supported in ActionScript 3.0. Instead, you can specify depth as integer depth value, or rewrite your code using ActionScript 3.0 Sprites instead of MovieClips.
`setMask()`	Target paths are not supported in ActionScript 3.0. Instead, you can specify the mask as a DisplayObject, or rewrite your code using ActionScript 3.0 Sprites instead of MovieClips.
`loadMovie()`	The *method* (GET or POST) optional argument to the `loadMovie()` method is not supported in ActionScript 3.0.
`getURL()`	The `getURL()` method has been deprecated. You should instead use the `navigateToURL()` method. For more information, see the *Flex 2 Developer's Guide*.
`loadMovieNum()`	The `loadMovieNum()` method is no longer supported in ActionScript 3.0. To achieve similar functionality, rewrite your code to use ActionScript 3.0 Sprites instead of ActionScript 2.0 MovieClips.
`loadVariables()`	The `loadVariables()` method is not supported in ActionScript 3.0.

The drawing methods such as `moveTo()` are now accessed via the `graphics` property of the MovieClip object. To use them you must import the flash.display.* package.

Arrays

Arrays no longer automatically update if you change the data. This means that you should no longer databind to an Array unless you manually call the `dispatchEvent()` each time an element in the Array changes. Otherwise, the control to which the data is bound will not reflect the changes. You can databind to a Collection.

Two convenience methods of the Array class have been removed. You can no longer use the `addItem()` and `removeItem()` methods. You must instead use `push()` and `pop()`, respectively. Array was formerly a mixin, but is now used directly from the Flash package without modification. For example:

```
var cards:Array;
// Old way.
cards.addItem({label: "Visa", data: CreditCardValidatorCardType.VISA});
// New way.
cards.push({label: "Visa", data: CreditCardValidatorCardType.VISA});
```

Arrays are commonly used as data providers in Flex applications.

setInterval() and clearInterval()

The `setInterval()` and `clearInterval()` methods are now in the flash.util package, so to use them you must import that package. But the `setInterval()` and `clearInterval()` methods are deprecated in favor of the new Timer class.

For an example that uses a timer in a Flex application, see "Using Timer" on page 182.

For information on using the Timer class, see the *ActionScript 3.0 Language Reference*.

Metadata

You must now separate metadata properties with commas. For example:

ActionScript 2.0:

```
[Style(name="horizontalAlign" enumeration="left,center,right"
  inherit="no")]
```

ActionScript 3.0:

```
[Style(name="horizontalAlign", enumeration="left,center,right",
  inherit="no")]
```

Constants

You can use the `const` keyword to define constants in ActionScript 3.0. First, you determine which class and instance variables qualify as constants, and then declare them with `const` instead of `var`; for example:

```
static const NONMODAL:Number = Alert.NONMODAL;
const backgroundColorName:String = "buttonColor";
```

In general, constants should be class constants rather than instance constants.

The initial value for a constant must be an expression that can be evaluated at compile time. Also, you cannot create constants of type Array or Object.

Method signatures

This section describes changes to method signatures in ActionScript 3.0.

No arguments

If a function takes no arguments, be sure to specify its argument list as `()` and not as `(void)`. The latter specifies a single argument of type Object named "void". Consider which, if any, of the arguments should be optional, and assign default values for them; for example:

```
override public function createClassObject(type:Class, name:String=null,
  depth:int=0, initObj:Object=null):UIObject
```

Variable number of arguments

If the function takes a variable number of arguments, use the new "..." syntax:

```
function function_name([ arguments ], ... arrayOfArgs)
```

For example:

```
function foo(n:Number, ... arrayOfArgs):void
```

The ". . ." and its array must be the last argument in the method. You cannot specify a type because it is always an Array containing the other arguments.

The following example shows how to use a method with a variable number of arguments. If you define the following:

```
function myfunc(arg1, ... arrayOfArgs)
```

you can call it as:

```
myfunc(a, b, c, d);
```

and the Array arrayOfArgs will have the value `[ b, c, d ]`.

__proto__

ActionScript 3.0 does not support "hacking" the prototype chain. The use of `__proto__` is no longer supported. For example:

ActionScript 2.0:

```
Class A {}
var a: A = new A;
trace(a.b)                 // Output: undefined
a.__proto__.b = 10         // Ok
trace(a.b)                 // Output: 10
class C {
   var x:Number = 20;
}
var c:C = new C();
a.__proto__ = C.prototype;
trace(a.x);                // Output: 20
```

ActionScript 3.0:

```
class A {}
var a: A = new A
trace(a.b)                      // Output: undefined
a.__proto__.b = 10              // Error, __proto__ unsupported.
                                // Use .constructor.prototype instead.
a.constructor.prototype.b = 10; // Ok
trace(a.b)                      // Output: 10
class C {
  var x:Number = 20;
}
var c:C = new C();
a.__proto__ = C.prototype;      // Error, __proto__ unsupported.
                                // .constructor.prototype is equivalent,
                                // though read only.
```

Primitive types

Primitive types are sealed classes and do not have object wrappers. Primitive types in ActionScript 3.0 are final, sealed classes. Furthermore, there are no object wrappers for primitives. As a result, you cannot create properties on them at run time; for example:

ActionScript 2.0:

```
newstring = new String("hello");
newstring.sayHi = function() {
  trace("hi!");
}
newstring.sayHi();
newstring2 = "hello";
```

```
newstring2.prop = 1;
trace(newstring2.prop);      // prints '1'
```

ActionScript 3.0:

```
newstring = new String("hello");
String.prototype.sayHi = function() {
  trace("hi!");
}
newstring.sayHi();
newstring2 = "hello";
newstring2.prop = 1;         // Warning, String is sealed...
String.prototype.prop = 1;
trace(newstring2.prop);      // Output: 1
```

Working with keys

The Key class is now the Keyboard class.

doLater() method

The `doLater()` method on UIComponent has been removed. You now use `callLater()`. The `callLater()` method takes a similar number of arguments. You no longer specify the object on which the function to be called later is defined. In addition, because of stricter typing, you must remove quotes from around the function if you used them; the type is function and cannot be coerced from type String. For example:

Flex 1.5:

```
doLater(this, "moveText");
```

Flex 2:

```
callLater(moveText);
```

The `callLater()` method still takes an optional `args` argument. For more information, see *ActionScript 3.0 Language Reference*.

LocalConnection

The `LocalConnection.allowDomain()` method specifies one or more domains that can send LocalConnection calls to this LocalConnection instance. In previous versions of ActionScript, the `allowDomain()` method was a callback method that you implemented. In ActionScript 3.0, `allowDomain()` is a built-in method of LocalConnection that you call. This makes the `allowDomain()` method work in much the same way as the `Security.allowDomain()` method.

CHAPTER 3

Flex Classes

3

This topic describes API changes to the Flex class library in Adobe Flex 2 SDK, including class-level changes to containers, core classes, and UI components.

Contents

Core classes

The UIObject and UIComponent classes have been combined. As a result, references to the UIObject class should be replaced by UIComponent. Visual Flex controls are subclasses of the UIComponent class (formerly UIObject). In Flex 1.5, the UIComponent class was a descendent of the MovieClip class. Now, the UIComponent class descends from the Sprite class. The following example shows the full hierarchy of a Flex control, such as the Button control:

```
Object
  |
  +--flash.events.EventDispatcher
    |
    +--flash.display.DisplayObject
      |
      +--flash.display.InteractiveObject
        |
        +--flash.display.FlashContainer
          |
          +--flash.display.Sprite
            |
            +--mx.core.UIComponent
              |
              +--mx.controls.Button
```

The EventDispatcher class was a mixin. It is now a base class in the flash.events package.

This section describes changes to Flex classes in the mx.core package.

mx.core.Application

The Application class now directly extends mx.core.Container instead of mx.containers.Box.

The following table describes changes to the mx.core.Application class:

Member	Change description
`alert()`	Removed. You must now use the `mx.controls.Alert.show()` method. For information about additional changes to Alert, see "mx.controls.Alert" on page 61.
`attachApplication`	Removed.
`backgroundColor`	For information on changes to setting the `backgroundColor` style property, see the entry for `fillAlphas`/`fillColors`.
`className`	Removed. Now on UIComponent.
`constructObject()`	Removed.

Member	Change description
`createLater`	`addToCreationQueue`
`direction`	The `direction` property has been replaced by the `layout` property.
`fillAlphas/fillColors`	The application background gradient now uses the `backgroundGradientAlphas` and `backgroundGradientColors` style properties for styling, instead of the `fillAlphas` and `fillColors` style properties. If the `backgroundGradientColors` property is undefined (which is the default), the background colors are calculated based on the `backgroundColor` style property. This means you no longer have to set the `fillColors` (or `backgroundGradientColors`) property when setting the `backgroundColor` property unless you require specific control over the gradient colors.
`getURL()`	Removed. For more information, see "getURL() method" on page 51.
`handleEvent()`	Removed.
`isFontEmbedded()`	Removed. Use the SystemManager's `isFontFaceEmbedded()` method instead.
`marginBottom`	`paddingBottom`
`marginLeft`	`paddingLeft`
`marginRight`	`paddingRight`
`marginTop`	`paddingTop`
`modalTransparency`	The range for this property is now 0 to 1, instead of 0 to 100.
`onSetFocus()`	Removed.
`resize()`	Removed. Now in the UIComponent class.
`selfContained`	Removed.

mx.core.Container

The class as been moved to the mx.core package.

The following table describes changes to the Container class:

Member	Change description
`allChildrenList`	`rawChildren`
`backgroundAlpha`	Deprecated. Use the `background` style to set this value.
`backgroundDisabledColor`	Deprecated.
`backgroundSize`	Deprecated. Use the `background` style to set this value.
`childAdded` (formerly `childCreated`)	`childAdd`
`childRemoved` (formerly `childDestroyed`)	`childRemove`
`childIndexChanged`	`childIndexChange`
`childrenCreationCompleteEffect`	Removed. Use the `creationCompleteEffect` effect.
`createComponent()`	`createComponentFromDescriptor()`. This method now takes a descriptor as its first argument rather than either a descriptor or a descriptor index. If you know the index, use `childDescriptors[index]` to get the descriptor itself. This method requires that you call the `validateNow()` method on the container to make the specified component appear in the display list. Alternatively, you can use the `createComponentsFromDescriptors()` method to create all components of the container.
`createComponents()`	`createComponentsFromDescriptors()`
`createdComponents`	This property is now internal only.
`dataObjectChanged`	`dataChange`
`defaultButton`	The `defaultButton` property of a container used to be set as follows: <mx:Form defaultButton="id"> You must now set it as follows: <mx:Form defaultButton="{id}"> Where `id` is the name of the Button control given in the MXML tag.
`dropShadow`	`dropShadowEnabled`
`hLineScrollSize`	`horizontalLineScrollSize`
`hPageScrollSize`	`horizontalPageScrollSize`
`hPosition`	`horizontalScrollPosition`
`hScrollBarStyleName`	`horizontalScrollBarStyleName`

Member	Change description
`hScroller`	`horizontalScroller`
`hScrollPolicy`	`horizontalScrollPolicy`
`marginBottom`	`paddingBottom`
`marginLeft`	`paddingLeft`
`marginRight`	`paddingRight`
`marginTop`	`paddingTop`
`maxHPosition`	`maxHorizontalScrollPosition`
`maxVPosition`	`maxVerticalScrollPosition`
`showInAutomationHierarchy`	Removed.
`viewMetricsAndMargins`	`viewMetricsAndPadding`
`vLineScrollSize`	`verticalLineScrollSize`
`vPageScrollSize`	`verticalPageScrollSize`
`vPosition`	`verticalSrollPosition`
`vScrollBarStyleName`	`verticalScrollBarStyleName`
`vScroller`	`verticalScroller`
`vScrollPolicy`	`verticalScrollPolicy`

If you set the `background` style, Flex ignores any value that is supplied for `backgroundAlpha`, `backgroundColor`, `backgroundImage`, or `backgroundSize`. If you do not specify a value for the `background` style, Flex uses the `backgroundAlpha`, `backgroundColor`, `backgroundImage`, and `backgroundSize` properties, as it did in Flex 1.5. Flex displays a deprecation warning if you use `backgroundAlpha` or `backgroundSize`.

mx.core.ContainerAllChildrenList

This class is now named ContainerRawChildrenList and is private.

mx.core.ContainerScrollPolicy

This class is now named ScrollPolicy.

mx.core.MovieClipLoaderAsset

The following table describes changes to the mx.core.MovieClipLoaderAsset class:

Member	Change description
`addedHandler`	Removed. Use the `complete` event instead.

mx.core.MXMLUIObject

The MXMLObject class is now the IMXMLObject class, to conform with the Flex 2 interface naming style.

mx.core.Repeater

The following table describes changes to the mx.core.Repeater class:

Member	Change description
`showInAutomationHierarchy`	Removed.

mx.core.ScrollView

The ScrollView class is now named ScrollControlBase.

The following table describes changes to the ScrollView class:

Member	Change description
`dropShadow`	`dropShadowEnabled`
`hPosition`	`horizontalScrollPosition`
`hScrollBarStyleName`	`horizontalScrollBarStyleName`
`hScroller`	`horizontalScroller`
`hScrollPolicy`	`horizontalScrollPolicy`
`maxHPosition`	`maxHorizontalScrollPosition`
`maxVPosition`	`maxVerticalScrollPosition`
`vPosition`	`verticalScrollPosition`
`vScrollBarStyleName`	`verticalScrollBarStyleName`
`vScroller`	`verticalScroller`
`vScrollPolicy`	`verticalScrollPolicy`

mx.core.Skin*

The mx.core.Skin* classes have been renamed to use the term Asset in the class name. For example, the SkinSprite class is now named SpriteAsset, and the SkinMovieClip class is now named MovieClipAsset.

These classes now implement the IFlexAsset interface.

mx.core.UIComponent

The UIComponent class has been combined with UIObject to form a single base class for visual Flex components. As a result, many of the changes listed in this section apply to properties, methods, and other members that were originally of the UIObject class.

The following table describes changes to the mx.core.UIComponent class:

Member	Change description
`automationComposite`	Removed.
`automationDelegate`	Removed.
`automationName`	Removed.
`automationParent`	Removed.
`automationValue`	Removed.
`childrenCreated()`	This method is now protected and not public. It is a component life-cycle method that the framework calls, and which component developers must override, but it should not be called directly.
`className`	Read-only.
`commitProperties()`	This method is now protected and not public. It is a component life-cycle method that the framework calls, and which component developers must override, but it should not be called directly.
`constructObject2()`	Removed. Use the `new` operator and the `addChild()` method or other methods to add new visual objects to the display list.
`createChildren()`	This method is now protected and not public. It is a component life-cycle method that the framework calls, and which component developers must override, but it should not be called directly.

Member	Change description
`createEmptyObject()`	Removed. Use the `new` operator and the `addChild()` method or other methods to add new visual objects to the display list.
`createToolTip`	`toolTipCreate`
`currentStateChanged`	`currentStateChange`
`deleteStyle()`	`clearStyle()`
`destroyObject()`	Removed. Use the `removeChild()` method or other methods to remove a child from its parent.
`doLater()`	`callLater()`
`draw`	`updateComplete`
`drawRect()`	Removed. Use the `drawRect()` method of the flash.display.Graphics class.
`endToolTip`	`toolTipEnd`
`fillRect()`	Removed. Use the fill methods of the flash.display.Graphics class.
`getFocusManager()`	`focusManager`
`getSystemManager()`	`systemManager`
`getUnscaledHeight()`	`unscaledHeight`
`getUnscaledWidth()`	`unscaledWidth`
`hideToolTip`	`toolTipHide`
`invalidateStyle()`	`styleChanged()`
`isChildOf()`	Removed. Use the `contains()` method of the DisplayObjectContainer class.
`isParentOf()`	Removed. Use the `contains()` method of the DisplayObjectContainer class.
`layoutComplete`	`updateComplete`
`MAX_HEIGHT`	`DEFAULT_MAX_HEIGHT`
`MAX_WIDTH`	`DEFAULT_MAX_WIDTH`
`measure()`	This method is now protected and not public. It is a component life-cycle method that the framework calls, and which component developers must override, but it should not be called directly.
`mouseDownOutside`	This event is now of type FlexMouseEvent instead of MouseEvent.

Member	Change description
`mouseOverEffect`	`rollOverEffect`
`mouseOutEffect`	`rollOutEffect`
`mouseWheelOutside`	This event is now of type `FlexMouseEvent` instead of `MouseEvent`.
`notifyEffectEnding`	`effectEnding`
`notifyEffectPlaying`	`effectPlaying`
`popUp`	`isPopUp`
`record`	This event is no longer dispatched by UIComponent.
`regenerateProtoChain()`	`regenerateStyleCache()`
`scrollTrackColor`	Removed.
`setFocusLater()`	This method is now private.
`setSize()`	`setActualSize()`
`showToolTip`	`toolTipShow`
`startToolTip`	`toolTipStart`
`strokeRoundRect()`	Removed. Use the `drawRoundRect()` method of the flash.display.Graphics class.
`themeColor`	The `themeColor` property is now a style property only. In Flex 1.5, it was a standard property of UIComponent as well as a style property. You can still apply `themeColor` in an MXML tag, but you must use the `getStyle()` and `setStyle()` methods to access this property from ActionScript.
`updateDisplayList()`	This method is now protected and not public. It is a component life-cycle method that the framework calls, and which component developers must override, but it should not be called directly.
`updateNow()`	`validateNow()`
`validationFailed`	`validationResultHandler()`
`validationSucceeded()`	`validationResultHandler()`
`valueCommitted`	`valueCommit`

The alpha, scaleX, and scaleY properties

NOTE: This change formerly applied to the UIObject class, but because UIObject and UIComponent have been combined into a single base class, the change now applies to the UIComponent class.

The `alpha`, `scaleX`, and `scaleY` properties now range from 0 to 1 instead of from 0 to 100. For example, to make a UIObject 50% opaque, specify `alpha=0.5` instead of `alpha=50`. To stretch the object horizontally by a factor of two, specify `scaleX=2` instead of `scaleX=200`.

The default value for the `focusAlpha` property is now 0.3.

The width and height properties

NOTE: This change formerly applied to the UIObject class, but because UIObject and UIComponent have been combined into a single base class, the change now applies to the UIComponent class.

The `width` and `height` properties are now typed as Number. In Flex 1.5, you could set the `width` and `height` to a Number (such as 50) or a String (such as "50%"). Because the type is now only a Number, you can no longer use the following syntax in your ActionScript:

```
myUIObject.width = "50%";
```

To set widths and heights using percentage values, you must now use the following syntax:

```
myUIObject.percentWidth = 50;
```

However, in MXML, you can express the `width` property as a pixel or percentage value. For example, the following two lines of code are valid:

```
<Canvas width="50%">
<Canvas percentWidth="50">
```

The addEventHandler() method

NOTE: This change formerly applied to the UIObject class, but because UIObject and UIComponent have been combined into a single base class, the change now applies to the UIComponent class.

The `addEventHandler()` method has been removed from UIObject. This method is now inherited from the flash.events.EventDispatcher class.

getURL() method

NOTE

This change formerly applied to the UIObject class, but because UIObject and UIComponent have been combined into a single base class, the change now applies to the UIComponent class.

In Flex 1.5, every UIObject inherited the `getURL()` method from the MovieClip class. This is no longer the case. You now use the `navigateToURL()` method in the flash.net package, which takes a URLRequest object rather than a String for the URL:

```
navigateToURL(new URLRequest('url'));
```

The load and unload events

NOTE

This change formerly applied to the UIObject class, but because UIObject and UIComponent have been combined into a single base class, the change now applies to the UIComponent class.

The `load` event, which was previously deprecated, has been removed.

The `unload` event has been moved to the SWFLoader object and is dispatched whenever a loaded SWF file is removed using the `LSWFLoader.unload()` method.

Validation and layout methods

The `invalidateLayout()`, `invalidate()`, `layoutChildren()` and `draw()` methods have been removed. Subclasses now override the `updateDisplayList()` method, which is now protected instead of public.

Drawing methods

NOTE

This change formerly applied to the UIObject class, but because UIObject and UIComponent have been combined into a single base class, the change now applies to the UIComponent class.

The Adobe Flash drawing API methods, such as `beginFill()` and `moveTo()`, are no longer inherited by UIObject; they are now methods of the Graphics object, which you access with the `graphics` property of Sprite, which UIObject inherits.

Enumerated values are now constants

In many cases, properties that took a predefined list of values (such as the Button control's `labelPlacement` property taking `right`, `left`, `bottom`, and `top`) now take class constants as well. For example, the `labelPlacement` property can now take one of the following constants:

- `ButtonLabelPlacement.RIGHT`
- `ButtonLabelPlacement.LEFT`
- `ButtonLabelPlacement.BOTTOM`
- `ButtonLabelPlacement.TOP`

Other properties that now use constants include the Box control's `direction` property; the Container control's `creationPolicy`, `hScrollPolicy`, and `vScrollPolicy` properties, and the ProgressBar control's `direction`, `labelPlacement`, and `mode` properties.

This change applies to ActionScript code. In general, the MXML usage has not changed.

life-cycle methods

The `createChildren()`, `childrenCreated()`, `commitProperties()`, `measure()`, and `updateDisplayList()` methods of UIComponent are now protected rather than public. These are component life-cycle methods that the framework calls, and which component developers must override, but they should not be called directly.

Flex 1.x:

```
override public function measure():void {
   ...
}
```

Flex 2:

```
override protected function measure():void {
   ...
}
```

mx.core.UIObject

The UIObject base class has been merged with the UIComponent class. UIComponent is now the lowest-level base class for nonskins. For information about changes to the combined object, see "mx.core.UIComponent" on page 47.

mx.core.UITextFormat

The following table describes the changes to the UITextFormat class:

Member	Change description
`isFontFaceEmbedded`	Moved to the SystemManager class.

mx.core.View

The mx.core.View class has been removed and its functionality has been distributed between the SWFLoader (formerly Loader), ScrollControlBase (formerly ScrollView), and Container classes.

Object creation and destruction methods that were on the View class have been removed. To create a new object, you now use the `new` operator and use the `addChild()` method or other methods to add the new object to the parent container. To destroy an object, you use the `removeChild()` or other methods to remove the object from the parent container.

The following methods of the View class have been removed:

- `createChild()`
- `createChildWithStyles()`
- `destroyChild()`
- `destroyChildAt()`
- `destroyAllChildren()`

Containers

This section describes changes to Flex classes in the mx.containers package. In addition to the changes listed here, there are also changes to the layouts. For more information, see Chapter 15, “Using Layout Containers,” in *Flex 2 Developer’s Guide*.

mx.containers.Accordion

The following table describes changes to the Accordion class:

Member	Change description
Child indices	The types have been changed from uint to int.
`change`	Dispatched when a user clicked on an Accordion header or when you programmatically set the value of the `selectedIndex` property to a new number. Now the Accordion container dispatches a `change` event only when a button is pressed. The Accordion container dispatches a `valueCommit` event in both cases.
`changeEffect`	Use the `showEffect` and `hideEffect` effect triggers of children of the Accordion container.
`createSegment()`	Removed. Use the `new` operator and the `addChild()` method or other methods to add new visual objects to the display list.
`getHeaderAt`	Parameter renamed from `i` to `index`.
`headerClass`	`headerRenderer` (now typed IFactory)
`headerStyle`	`headerStyleName`
`historyManagement`	`historyManagementEnabled`
`newValue`	`newIndex`
`openEasing`	`openEasingFunction`
`marginBottom`	`paddingBottom`
`marginTop`	`paddingTop`
`prevValue`	`oldIndex`
`selectionChange`	`change`

mx.containers.accordionclasses.*

The accordionclasses package is now named accordionClasses.

mx.containers.ApplicationControlBar

The following table describes changes to the ApplicationControlBar class:

Member	Change description
`borderStyle`	Can no longer be set on ApplicationControlBar.
`fillAlphas`	The default value has changed to `[0,0]`.
`fillColor`	Is now `fillColors` (with an s). The default value is `[0xFFFFFF, 0xFFFFFF]`.

mx.containers.Box

The following table describes the changes to the Box class:

Member	Change description
`marginBottom`	`paddingBottom`
`marginTop`	`paddingTop`

mx.containers.buttonbarclasses.ButtonBarButton

This class has been moved to the mx.controls.buttonBarClasses package and is now private.

mx.containers.Canvas

The following table describes the changes to the Canvas class:

Member	Change description
`horizontalGap`	Removed.
`verticalGap`	Removed.

mx.containers.ControlBar

The ControlBar class now derives from mx.containers.Box instead of mx.containers.HBox.

The following table describes changes to the ControlBar class:

Member	Change description
`backgroundSkin`	Removed. This style property was not used.

mx.containers.DividedBox

The following table describes changes to the DividedBox class:

Member	Change description
`dividerDragged`	`dividerDrag`
`dividerPressed`	`dividerPress`
`dividerReleased`	`dividerRelease`
`getDividerCount`	`numDividers`
`horizontalCursor`	`horizontalDividerCursor`
`verticalCursor`	`verticalDividerCursor`

mx.containers.dividedboxclasses.*

The dividedboxclasses package is now named dividedBoxClasses.

mx.containers.Form

The following table describes changes to the Form class:

Member	Change description
`marginBottom`	`paddingBottom`
`marginTop`	`paddingTop`

mx.containers.FormHeading

The following table describes changes to the FormHeading class:

Member	Change description
`horizontalGap`	Removed.
`verticalGap`	`paddingTop`

mx.containers.FormItem

The following table describes changes to the FormItem class:

Member	Change description
`labelObject`	This property is now internal only.
`marginBottom`	`paddingBottom`
`marginRight`	`paddingRight`
`marginTop`	`paddingTop`

mx.containers.gridclasses.*

The gridclasses package is now named dataGridClasses.

mx.containers.GridRow

The GridRow class now subclasses HBox instead of Box.

mx.containers.HBox

The following table describes changes to the HBox class:

Member	Change description
`direction`	Removed.

mx.containers.HDividedBox

The following table describes changes to the HDividedBox class:

Member	Change description
`direction`	Removed.

mx.containers.LinkBar

The mx.containers.LinkBar class has been moved to mx.controls.LinkBar.

The following table describes changes to the LinkBar class:

Member	Change description
`click`	`itemClick`
`marginBottom`	`paddingBottom`
`marginTop`	`paddingTop`
`strokeColor`	`separatorColor`
`strokeWidth`	`separatorWidth`

mx.containers.NavBar

The mx.containers.NavBar class has been moved to mx.controls.NavBar.

The following table describes changes to the NavBar class:

Member	Change description
`click`	`itemClick`
"none selected"	The special value for "none selected" is -1 instead of `NaN`.
`selectedIndex`	The property in NavBar containers (including TabBar and LinkBar containers) is now public and is now an int instead of a Number. It is also now bindable.

mx.containers.Panel

Panel containers now extend mx.core.Container instead of mx.containers.Box.

The following table describes changes to the Panel class:

Member	Change description
`borderStyle`	The Panel container now supports the `borderStyle` property inherited from Container. In Flex 1.5, `borderStyle` was ignored. You could approximate the same behavior by setting the `borderThickness` property.
`controlBar`	This property is now protected.
`direction`	Replaced by the `layout` property.
`dropShadow`	`dropShadowEnabled`
`marginBottom`	`paddingBottom`
`marginTop`	`paddingTop`

Member	Change description
`modalTransparency`	The range for this property is now 0 to 1, instead of 0 to 100.
`panelAlpha`	`borderAlpha`
`panelBorderStyle`	`roundBottomCorners`
`statusStyleDeclaration`	Removed. Use the `statusStyleName` style.
`statusTestField`	This property is now protected.
`titleStyleDeclaration`	Removed. Use the `titleStyleName` style.
`titleTestField`	This property is now protected.

mx.containers.TabBar

The mx.containers.TabBar class has been moved to the mx.controls package. The TabBar container now extends mx.containers.ToggleButtonBar instead of mx.containers.NavBar.

The following table describes changes to the TabBar class:

Member	Change description
`activeTabStyleDeclaration`	`selectedTabTextStyleName`
`click`	`itemClick`
`tabSkin`	`tabStyleName`
`verticalAlign`	The default value is now `middle`. Previously, it was `top`, as inherited from the Box control.
`verticalGap`	The default value is now -1 (was 6, as inherited from the Box control).

mx.containers.tabbarclasses.Tab

This class has been moved to the mx.controls.tabBarClasses package and is now private.

mx.containers.TabNavigator

The following table describes changes to the TabNavigator class:

Member	Change description
`activeTabStyleDeclaration`	`selectedTabTextStyleName`
`createTab()`	Removed. Use the `addChild()` method.

Member	Change description
`getTabAt`	Parameter renamed from `i` to `index`.
`tabSkin`	`tabStyleName`

mx.containers.Tile

The following table describes changes to the Tile class:

Member	Change description
`marginBottom`	`paddingBottom`
`marginTop`	`paddingTop`

mx.containers.TitleWindow

The following table describes changes to the TitleWindow class:

Member	Change description
`closeButton`	`showCloseButton`

In Flex 1.x, clicking on the close button (and only the close button) generated a `click` event. In Flex 2, the `click` event is triggered when the user presses the mouse button anywhere in the TitleWindow area. You should change your `click` event handler to handle a `close` event instead.

mx.containers.ToggleButtonBar

The mx.containers.ToggleButtonBar class has been moved to the mx.controls package.

The following table describes changes to the ToggleButtonBar class:

Member	Change description
`selectedButtonTextStyleName`	`selectedItemStyleName`
`unselectable`	`alwaysToggleOnClick`

mx.containers.ViewStack

The following table describes changes to the ViewStack class:

Member	Change description
Child indices	The type of the child indices in the ViewStack container has been changed from uint to int.
`cachePolicy`	The ViewStack container no longer overrides the `cachePolicy` property.
`changeEffect`	Use the `showEffect` and `hideEffect` effect triggers of children of the ViewStack container instead.
`historyManagement`	`historyManagementEnabled`
`marginBottom`	`paddingBottom`
`marginTop`	`paddingTop`

When all children are removed from a ViewStack, Flex sets the `selectedIndex` to -1 (was `NaN`). Removing children from a ViewStack container now adjusts the `selectedIndex`.

If a view is removed from the front (where 0 is the front-most), Flex decrements the `selectedIndex` by 1 to remain pointing at the selected item. If the current view is removed, Flex points the `selectedIndex` to the next view first, and then the previous view.

If there are no remaining views, Flex sets the `selectedIndex` property to -1.

Controls

This section describes changes to Flex classes in the mx.controls package.

mx.controls.Alert

The following table describes changes to the Alert class:

Member	Change description
`buttonStyleDeclaration`	Removed. Use the `buttonStyleName` style.
`messageStyleDeclaration`	Removed. Use the `messageStyleName` style.

Member	Change description
`show()`	The `show()` method now has the following signature: `show(text:String, title:String=null, flags:uint=0x4, parent:Sprite=null, closeHandler:Function=null, iconClass:Class=null,defaultButtonFlag:uint=0x4):Alert`
`titleStyleDeclaration`	Removed. Use the `titleStyleName` style.

mx.controls.alertclasses.*

The alertclasses package is now named alertClasses.

mx.controls.Button

The Button class no longer inherits from SimpleButton.

The following table describes changes to the Button class:

Member	Change description
`buttonDragOut`	The Button control no longer dispatches this event.
`cornerRadius`	Removed.
`dataChanged`	`dataChange`
`falseDisabledSkin`	`disabledSkin`
`falseDownSkin`	`downSkin`
`falseOverSkin`	`overSkin`
`falseUpSkin`	`upSkin`
`icon`	Now a style of type Class not a property of type Object.
`marginBottom`	`paddingBottom`
`marginLeft`	`paddingLeft`
`marginRight`	`paddingRight`
`marginTop`	`paddingTop`
`selected`	Cannot be set if `toggle` is `false`. Flex now forces `selected` to `false` when `toggle` is `false`.
`trueDisabledSkin`	`selectedDisabledSkin`
`trueDownSkin`	`selectedDownSkin`
`trueOverSkin`	`selectedOverSkin`

Member	Change description
`trueUpSkin`	`selectedUpSkin`
`version`	Removed.

mx.controls.ButtonBar

The following table describes changes to the ButtonBar class:

Member	Change description
`click`	`itemClick`

mx.controls.ButtonBarButton

The ButtonBarButton class has been made internal only.

mx.controls.CalendarLayout

The following table describes changes to the CalendarLayout class:

Member	Change description
`background`	This property is now internal only.
`backMonthButton`	This property is now internal only.
`backMonthHit`	This property is now internal only.
`border`	This property is now internal only.
`callHeader`	This property is now internal only.
`disjointSelection`	`allowDisjointSelection`
`downYearButton`	This property is now internal only.
`downYearHit`	This property is now internal only.
`fwdMonthButton`	This property is now internal only.
`fwdMonthHit`	This property is now internal only.
`headerDisplay`	This property is now internal only.
`headerStyleDeclaration`	`headerStyleName`
`monthDisplay`	This property is now internal only.
`multipleSelection`	`allowMultipleSelection`
`todayStyleDeclaration`	`todayStyleName`

Member	Change description
`upYearButton`	This property is now internal only.
`upYearHit`	This property is now internal only.
`weekDayStyleDeclaration`	`weekDayStyleName`
`yearDisplay`	This property is now internal only.

mx.controls.ColorPicker

The following table describes changes to the ColorPicker class:

Member	Change description
`closeEasing`	`closeEasingFunction`
`marginBottom`	`paddingBottom`
`marginLeft`	`paddingLeft`
`marginRight`	`paddingRight`
`marginTop`	`paddingTop`
`openEasing`	`openEasingFunction`

mx.controls.ComboBase

The following table describes changes to the ComboBase class:

Member	Change description
`border`	This property is now internal.
`downArrowButton`	This property is now internal.
`length`	Removed.
`selectionChanged`	This property is now internal.
`selectedIndexChanged`	This property is now internal.
`selectedItemChanged`	This property is now internal.
`textInput`	This property is now internal.

mx.controls.ComboBox

The following table describes changes to the ComboBox class:

Member	Change description
`alternatingRowColors`	`alternatingItemColors`
`cellRenderer`	`itemRenderer`
`dataChanged`	`dataChange`
`textDisabledColor`	`disabledColor`
`itemSkin`	`itemRenderer`
`openEasing`	`openEasingFunction`
`selectionEasing`	`selectionEasingFunction`

mx.controls.DataGrid

The DataGrid class now extends the new GridBase class, which extends the ListBase class.

The following table describes changes to the DataGrid class:

Member	Change description
`addColumn()`	Removed.
`addColumnAt()`	Removed.
`cellBeginEdit`	`itemEditBegin`
`cellEditor`	`itemEditorInstance`
`cellEndEdit`	`itemEditEnd`
`cellFocusIn`	`itemFocusIn`
`cellFocusOut`	`itemFocusOut`
`cellPress`	Removed.
`cellRenderer`	`itemRenderer`
`cellRequestEdit`	`itemEditBeginning`
`columnNames`	Removed.
`editedCell`	`editedItemRenderer`
`focusedCell`	`editedItemPosition`
`getColumnAt()`	Removed.
`getColumnIndex()`	Removed.
`getColumnName()`	Removed.

Member	Change description
`headerColor`	`headerColors`
`headerStyle`	`headerStyleName`
`hGridLineColors`	`horizontalGridLineColors`
`hGridLines`	`horizontalGridLines`
`hPosition`	`horizontalScrollPosition`
`minColWidth`	`minColumnWidth`
`removeAllColumns()`	Removed.
`removeColumnAt()`	Removed.
`setColumnIndex`	Removed.
`vGridLineColors`	`verticalGridLineColors`
`vGridLines`	`verticalGridLines`

In Flex 2, when a DataGrid control's width is not wide enough to show all columns, only the first column gets smaller. This applies to DataGrid controls with a horizontal scroll policy of `true` or `auto`. In Flex 1.5, DataGrid controls tried to squeeze in all the columns ignoring the their `minColWidth` property.

mx.controls.dataGridClasses.DataGridColumn

The following table describes changes to the DataGridColumn class:

Member	Change description
`cellRenderer`	`itemRenderer`
`columnName`	`dataField`
`editorClass`	`itemEditor`
`editorProperty`	`editorDataField`
`headerClass`	`headerRenderer` (now typed IFactory)
`headerStyle`	`headerStyleName`
`itemSkin`	`itemRenderer`
`styleName`	Use individual style properties on DataGridColumn.

mx.controls.dataGridClasses.DataGridListData

The following table describes changes to the DataGridListData class:

Member	Change description
`columnName`	`dataField`

mx.controls.DateChooser

The following table describes changes to the DateChooser class:

Member	Change description
`background`	This property is now internal only.
`backMonthButton`	This property is now internal only.
`backMonthHit`	This property is now internal only.
`border`	This property is now internal only.
`callHeader`	This property is now internal only.
`disjointSelection`	`allowDisjointSelection`
`downYearButton`	This property is now internal only.
`downYearHit`	This property is now internal only.
`fwdMonthButton`	This property is now internal only.
`fwdMonthHit`	This property is now internal only.
`headerColor`	`headerColors`
`headerDisplay`	This property is now internal only.
`headerStyle`	`headerStyleName`
`headerStyleDeclaration`	`headerStyleName`
`horizontalGap`	Removed.
`monthDisplay`	This property is now internal only.
`multipleSelection`	`allowMultipleSelection`
`todayStyleDeclaration`	`todayStyleName`
`upYearButton`	This property is now internal only.
`upYearHit`	This property is now internal only.
`verticalGap`	Removed.

Member	Change description
`weekDayStyleDeclaration`	`weekDayStyleName`
`yearDisplay`	This property is now internal only.

mx.controls.DateField

The following table describes changes to the DateField class:

Member	Change description
`dataChanged`	`dataChange`
`formattingFunction`	`labelFunction`
`headerStyleDeclaration`	`headerStyleName`
`headerColor`	Removed. Replaced with the `dateChooserStyleName` property.
`headerColors`	Deprecated. Replaced with the `dateChooserStyleName` property.
`headerStyle`	`headerStyleName`
`parsingFunction`	`parseFunction`
`pulldown`	`dropdown`
`rollOverColor`	Deprecated. Replaced with the `dateChooserStyleName` property.
`selectionColor`	Deprecated. Replaced with the `dateChooserStyleName` property.
`todayColor`	Deprecated. Replaced with the `dateChooserStyleName` property.
`todayStyleDeclaration`	`todayStyleName`
`weekDayStyleDeclaration`	`weekDayStyleName`

mx.controls.HorizontalList

The following table describes changes to the HorizontalList class:

Member	Change description
`cellRenderer`	`listItemRenderer`
`itemWidth`	`columnWidth`

mx.controls.HRule

The following table describes changes to the HRule class:

Member	Change description
`color`	`strokeColor`

mx.controls.HSlider

The following table describes changes to the HSlider class:

Member	Change description
`labelStyleDeclaration`	Removed. Use the `labelStyleName` style instead.
`showTicks`	Removed. Instead of setting it to `false` to turn off tick marks, you set `tickInterval` to 0; for example: `<mx:HSlider id="hs1" snapInterval="1" tickInterval="0"/>`
`toolTipStyleDeclaration`	Removed. Use the `toolTipStyleName` style instead.

mx.controls.Image

The following table describes changes to the Image class:

Member	Change description
`dataChanged`	`dataChange`

mx.controls.Label

The Label class now extends mx.core.UIComponent rather than mx.core.UIObject.

The following table describes changes to the Label class:

Member	Change description
`dataChanged`	`dataChange`

mx.controls.Link

The Link class is now named LinkButton.

mx.controls.List

The following table describes changes made to the List class:

Member	Change description
`cellBeginEdit`	`itemEditBegin`
`cellEditor`	`itemEditorInstance`
`cellEndEdit`	`itemEditEnd`
`cellFocusIn`	`itemFocusIn`
`cellFocusOut`	`itemFocusOut`
`cellRequestEdit`	`itemEditBeginning`
`editedCell`	`editedItemRenderer`
`focusedCell`	`editedItemPosition`
`getItemAt()`	DataProvider APIs are no longer on the list-based classes. Instead of `myList.getItemAt(index)`, you use `myList.dataProvider.getItemAt(index)`.
`isCellEditor`	`rendererIsEditor`

mx.controls.listclasses.*

The listclasses package name has been changed to listClasses.

The class hierarchy for the list-based classes has changed, as follows:

```
ListBase
  +-GridBase
    +-DataGrid
  +-List
    +-Menu
    +-Tree
  +-TileBase
    +-HorizontalList
    +-TileList
```

A `dragManager` property has been added to the List classes. If you are using the drag-and-drop operation in the List classes, you must set `dragManager="DragManager"` in order for the drag-and-drop operation to work properly.

mx.controls.listclasses.DataProvider

The DataProvider class is removed. Use Collections instead.

For more information, see Chapter 4, “Data Providers,” on page 101.

mx.controls.listclasses.ListBase

The following table describes changes to the ListBase class:

Member	Change description
`activeTabStyleDeclaration`	`SelectedTabStyleName`
`alternatingRowColors`	`alternatingItemColors`
`cachedPaddingBottom`	This property is now internal.
`cachedPaddingTop`	This property is now internal.
`cachedVerticalAlign`	This property is now internal.
`calculateHeight`	`measureHeightOfItems`
`calculateWidth`	`measureWidthOfItems`
`cellRenderer`	`itemRenderer`
`commitSelectedIndex()`	This method is now internal.
`commitSelectedIndices()`	This method is now internal.
`dataChanged`	`dataChange`
`defaultIcon`	Removed.
`getItemRendererForData()`	`itemToItemRenderer()`
`isHighLighted`	`isItemHighlighted`
`isSelected`	`isItemSelected`
`itemSkin`	`itemRenderer`
`itemToString`	`itemToLabel`
`listItemRenderer`	`itemRenderer`
`marginBottom`	`paddingBottom`
`marginLeft`	`paddingLeft`
`marginRight`	`paddingRight`
`marginTop`	`paddingTop`
`multipleSelection`	`allowMultipleSelection`
`selectionEasing`	`selectionEasingFunction`
`setColumnCount()`	This method is now internal.
`setColumnWidth()`	This method is now internal.
`textDisabledColor`	Removed.

mx.controls.listClasses.ListCellRenderer

The ListCellRenderer class is now named ListItemRenderer.

The following table describes changes to the ListItemRenderer class:

Member	Change description
`dataChanged`	`dataChange`

mx.controls.listClasses.TileListItemRenderer

The following table describes changes to the TileListItemRenderer class:

Member	Change description
`dataChanged`	`dataChange`

mx.controls.Loader

The mx.controls.Loader class name has changed to mx.controls.SWFLoader.

The following table describes changes to the Loader class:

Member	Change description
`border`	Removed.
`borderMetrics`	Removed.
`brokenImage`	The `brokenImage` is now the `brokenImageSkin` style property. This style property is of type Class, as are all other skin style properties.
`contentPath`	`source`
`load()`	Using the `load()` method to load a JPEG, GIF, or PNG file now creates an ImageSprite (containing an Image) as the child of the SWFLoader.
Styles	You can no longer set border and background color styles on the SWFLoader control.

Adobe Flash Player 9 now dispatches `ioError` events when external data such as the image cannot be loaded. The external loading code in View now listens for `ioError` events and `Error` events and forwards them. The SWFLoader class traps these events and displays the broken image.

mx.controls.Menu

The following table describes changes to the Menu class:

Member	Change description
`alternatingRowColors`	`alternatingItemColors`
`cellRenderer`	`listItemRenderer`
`change`	`itemClick`
`defaultIcon`	Removed.
`getMenuItemAt()`	Removed.
`menuData`	This property is now internal.
`menuItemRollOut`	`itemRollOut`
`menuItemRollOver`	`itemRollOver`
`popupDuration`	`openDuration`
`setMenuItemSelected()`	This method has been made protected.
`rootVisible`	`showRoot`
`textDisabledColor`	`disabledColor`

mx.controls.MenuBar

The following table describes changes to the MenuBar class:

Member	Change description
`change`	`itemClick`
`getMenuBarItemAt()`	Removed.
`labels`	Removed.
`menuItemRollOut`	`itemRollOut`
`menuItemRollOver`	`itemRollOver`
`rootModel`	This property is now internal.
`rootVisible`	`showRoot`
`selectionDisabledColor`	Removed.
`textDisabledColor`	`disabledColor`
`textRollOverColor`	Removed.
`textSelectedColor`	Removed.
`useRollOver`	Removed.

mx.controls.menuclasses.*

The menuclasses package has been renamed menuClasses.

mx.controls.menuclasses.IMenuDataDescriptor

The following table describes changes to the IMenuDataDescriptor class:

Member	Change description
`isSelected()`	`isToggled()`
`setSelected()`	`setToggled()`

mx.controls.menuclasses.MenuCellRenderer

The MenuCellRenderer class is now named MenuItemRenderer.

The following table describes changes to the MenuItemRenderer class:

Member	Change description
`dataChanged`	`dataChange`

mx.controls.NumericStepper

The following table describes changes to the NumericStepper class:

Member	Change description
`dataChanged`	`dataChange`
`dropShadow`	`dropShadowEnabled`
`trackSkin`	Removed.

mx.controls.PopUpButton

The following table describes changes to the PopUpButton class:

Member	Change description
`closePopUp()`	`close()`
`openEasing`	`openEasingFunction`
`openPopUp()`	`open()`

Member	Change description
`popUpObject`	`popUp`
`popUpOnMainButton`	`openAlways`

mx.controls.PopUpMenuButton

The following table describes changes to the PopUpMenuButton class:

Member	Change description
`change`	`itemClick`
`menuItemRollOut`	`itemRollOut`
`menuItemRollOver`	`itemRollOver`
`popUpObject`	`popUp`

mx.controls.RadioButton

The following table describes changes to the RadioButton class:

Member	Change description
`data`	`value`

mx.controls.RadioButtonGroup

The following table describes changes to the RadioButtonGroup class:

Member	Change description
`click`	`itemClick`
`selectedData`	`selectedValue`

mx.controls.RichTextEditor

The following table describes changes to the RichTextEditor class:

Member	Change description
`controlBarVisible`	`showControlBar`
`enableToolTip`	`showToolTips`
`selectedTextRange`	`selection`

mx.controls.richtexteditorclasses.*

The richtexteditorclasses package is now named richTextEditorClasses.

mx.controls.scrollClasses.ScrollBar

The following table describes changes to the ScrollBar class:

Member	Change description
`maxPos`	`maxScrollPosition`
`minPos`	`minScrollPosition`
`thumbDisabledSkin`	Removed.

mx.controls.SimpleButton

The SimpleButton class has been removed.

mx.controls.Slider

The thumb labels, which were known as ToolTips on the Slider, HSlider, and VSlider controls, are now known as data tips on those controls. Several property names have changed to reflect this change. The controls now also support traditional ToolTip labels, which are labels that appear for the entire control. You set these by using the inherited `toolTip` property.

The following table describes changes to the Slider class:

Member	Change description
`labelStyleDeclaration`	Removed. Use the `labelStyleName` style.
`maxValue`	`maximum`
`minValue`	`minimum`
`scrollTrackHeight`	Removed.
`showTicks`	Removed. Instead of setting it to `false` to turn off tick marks, you set `tickInterval` to 0; for example: `<mx:HSlider id="hs1" snapInterval="1" tickInterval="0"/>`
`slideEasing`	`slideEasingFunction`
`showValueTip`	`showDataTip`
`sliderToolTipClass`	`sliderDataTipClass`

Member	Change description
`snapToTicks`	`snapInterval`
`thumbDragged`	`thumbDrag`
`thumbPressed`	`thumbPress`
`thumbReleased`	`thumbRelease`
`thumbWidth`	Removed. Use thumb skins to create custom thumbs.
`tickFrequency`	`tickInterval`
`tickHeight`	`tickLength`
`tickSpacing`	`tickOffset`
`toolTipFormatFunction`	`dataTipFormatFunction`
`toolTipOffset`	`dataTipOffset`
`toolTipPlacement`	`dataTipPlacement`
`toolTipPrecision`	`dataTipPrecision`
`toolTipStyleDeclaration`	Removed. Use the `toolTipStyleName` style.
`toolTipStyleName`	`dataTipStyleName`
`trackHighlight`	`showTrackHighlight`

mx.controls.sliderclasses.*

The sliderclasses package is now named sliderClasses.

mx.controls.sliderclasses.SliderToolTip

The SliderToolTip class is now named SliderDataTip.

mx.controls.TextArea

The following table describes changes to the TextArea class:

Member	Change description
`dataChanged`	`dataChange`
`hPosition`	`horizontalScrollPosition`
`hScrollPolicy`	`horizontalScrollPolicy`
`linkColor`	Removed.
`maxVPosition`	`maxVerticalScrollPosition`

Member	Change description
`maxHPosition`	`maxHorizontalScrollPosition`
`password`	`displayAsPassword`
`underlineLink`	Removed.
`vPosition`	`verticalSrollPosition`
`vScrollPolicy`	`verticalScrollPolicy`

mx.controls.textclasses.*

The textclasses package has been renamed textClasses.

mx.controls.TextInput

The following table describes changes to the TextInput class:

Member	Change description
`dataChanged`	`dataChange`
`dropShadow`	`dropShadowEnabled`
`hPosition`	`horizontalScrollPosition`
`maxHPosition`	`maxHorizontalScrollPosition`
`password`	`displayAsPassword`
`text`	You can now set the `text` property of a TextInput control to `null`.

mx.controls.TileList

The following table describes changes to the TileList class:

Member	Change description
`cellRenderer`	`listItemRenderer`
`itemHeight`	`rowHeight`
`itemWidth`	`columnWidth`

Flex 1.x:

```
<mx:TileList id="myTile" dataProvider="{dataObject}"
  cellRenderer="ProdtThumbnail" itemWidth="120" itemHeight="116">
```

Flex 2:

```
<mx:TileList id="myTile" dataProvider="{dataObject}"
  listItemRenderer="ProdThumbnail" columnWidth="120" rowHeight="116">
```

mx.controls.ToolTip

The following table describes changes to the ToolTip class:

Member	Change description
`dropShadow`	`dropShadowEnabled`

mx.controls.Tree

The following table describes changes to the Tree class:

Member	Change description
`addChildItem()`	This method is now internal.
`alternatingRowColors`	`alternatingItemColors`
`cellPress`	Removed.
`defaultIcon`	Removed.
`expandItemHandler()`	This method is now internal.
`getNodeDisplayedAt()`	Removed. Use `Tree.listItems[rowIndex][0].data` instead.
`maxHPosition`	`maxHorizontalScrollPosition`
`openEasing`	`openEasingFunction`
`removeChildItem()`	This method is now internal.
`rootCollectionChangedHandler()`	This method is now internal.
`rootVisible`	`showRoot`
`selectionDuration`	Removed.
`selectionEasing`	`selectionEasingFunction` Was of type Time; is now of type Function.
`textDisabledColor`	Removed.

mx.controls.treeclasses.*

The treeclasses package is now named treeClasses.

mx.controls.treeclasses.DefaultDataDescriptor

The following table describes changes to the DefaultDataDescriptor class:

Member	Change description
`isBranch()`	Supports nodes in E4X XML, but not XMLNode objects or data that is serialized into ActionScript objects.
`isSelected()`	`isToggled()`
`setSelected()`	`setToggled()`

mx.controls.treeClasses.TreeCellRenderer

The TreeCellRenderer class is now named TreeItemRenderer.

The following table describes changes to the TreeItemRenderer class:

Member	Change description
`dataChanged`	`dataChange`

mx.controls.VRule

The following table describes changes to the VRule class:

Member	Change description
`color`	`strokeColor`

mx.controls.VSlider

The following table describes changes to the VSlider class:

Member	Change description
`labelStyleDeclaration`	Removed. Use the `labelStyleName` style.

Member	Change description

`showTicks`	Removed. Instead of setting it to `false` to turn off tick marks, you set `tickInterval` to 0; for example:
	`<mx:HSlider id="hs1" snapInterval="1"`
	`   tickInterval="0"/>`

`toolTipStyleDeclaration`	Removed. Use the `toolTipStyleName` style.

Effects

The "instance" classes (such as mx.effects.RotateInstance and mx.effects.ResizeInstance) have been moved to the mx.effects.effectClasses package.

This section describes changes to individual effects classes. In addition to the changes listed here, the effects architecture has changed. For more information, see Chapter 8, "Behaviors," on page 129.

The `name` property is now `id`.

mx.effects.AnimateProperty

The following table describes changes to the AnimateProperty class:

Member	Change description
`endValue`	`toValue`
`startValue`	`fromValue`

mx.effects.Effect

The following table describes changes to the Effect class:

Member	Change description
`effectProperties`	`relevantProperties`
`effectStyles`	`relevantStyles`
`endEffect()`	`end()`
`listener`	Removed.

`playEffect()`	Use the `play()` method instead. The method signature has changed to the following:
	`play(`*`targets`*`:Array=null,`
	`   `*`playReversedFromEnd`*`:Boolean=false):Array`

`repeat`	`repeatCount`

mx.effects.EffectInstance

The following table describes changes to the EffectInstance class:

Member	Change description
`repeat`	`repeatCount`
`stopRepeat`	This property is now internal.

mx.effects.MaskEffect

The following table describes changes to the MaskEffect class:

Member	Change description
`moveEasing`	`moveEasingFunction`
`scaleEasing`	`scaleEasingFunction`

These changes also apply to the MaskEffectInstance class.

mx.effects.Resize

The following table describes changes to the Resize class:

Member	Change description
`hideChildren`	`hideChildrenTargets`

These changes also apply to the mx.effects.ResizeInstance class.

mx.effects.SetPropertyAction

The following table describes changes to the SetPropertyAction class:

Member	Change description
`property`	`name`

mx.effects.SetStyleAction

The following table describes changes to the SetPropertyAction class:

Member	Change description
`property`	`name`

mx.effects.SoundEffect

The following table describes changes to the SoundEffect class:

Member	Change description
`panEasing`	`panEasingFunction`
`soundHolder`	Renamed to the `sound` property. This property is now read-write.
`volumeEasing`	`volumeEasingFunction`

These changes also apply to the SoundEffectInstance class.

mx.effects.Tween

The following table describes changes to the Tween class:

Member	Change description
`easing`	`easingFunction`

mx.effects.TweenEffect

The following table describes changes to the TweenEffect class:

Member	Change description
`easing`	`easingFunction`
`tweenEndHandler`	`tweenEnd`

These changes also apply to the TweenEffectInstance class.

Events

All events in the mx.collections package were moved to the mx.events package.

In addition to the changes listed here, the events architecture has changed. For more information, see Chapter 6, “Events,” on page 111.

mx.events.ChildExistenceChangedEvent

The following table describes changes to the ChildExistenceChangedEvent class:

Member	Change description
`CHILD_ADDED`	`CHILD_ADD`
`CHILD_REMOVED`	`CHILD_REMOVE`

mx.events.CalendarLayoutChangeEvent

The following table describes changes to the CalendarLayoutChangeEvent class:

Member	Change description
`cause`	`triggerEvent`

mx.events.CollectionEvent

The following table describes changes to the CollectionEvent class:

Member	Change description
`modelChanged`	`collectionChange`

mx.events.CursorEvent

This class has been removed.

mx.events.DataGridCellRenderer

The DataGridCellRenderer class is now named DataGridItemRenderer.

The following table describes changes to the DataGridItemRenderer class:

Member	Change description
`dataChanged`	`dataChange`

mx.events.DataGridEvent

The following table describes changes to the DataGridEvent class:

Member	Change description
`cell`	`itemRenderer`
`CELL_BEGIN_EDIT`	`ITEM_EDIT_BEGIN`
`CELL_END_EDIT`	`ITEM_EDIT_END`
`CELL_FOCUS_IN`	`ITEM_FOCUS_IN`
`CELL_FOCUS_OUT`	`ITEM_FOCUS_OUT`
`CELL_REQUEST_EDIT`	`ITEM_EDIT_BEGINNING`
`CELL_PRESS`	Removed.
`cellRenderer`	`itemRenderer`
`columnName`	`dataField`
`itemIndex`	`rowIndex`
`itemSkin`	`itemRenderer`
`view`	Removed. Use `target` instead.

mx.events.DateChooserEvent

The following table describes changes to the DateChooserEvent class:

Member	Change description
`cause`	`triggerEvent`

mx.events.DividerEvent

The following table describes changes to the DividerEvent class:

Member	Change description
`DIVIDER_DRAGGED`	`DIVIDER_DRAG`
`DIVIDER_PRESSED`	`DIVIDER_PRESS`
`DIVIDER_RELEASED`	`DIVIDER_RELEASE`

mx.events.DropdownEvent

The following table describes changes to the DropdownEvent class:

Member	Change description
`inputType`	`triggerEvent`

mx.events.EventDispatcher

This class has been moved to flash.events.EventDispatcher and is no longer used as a mixin.

mx.events.FlexEvent

The following table describes changes to the FlexEvent class:

Member	Change description
`DATA_CHANGED`	`DATA_CHANGE`
`DATA_OBJECT_CHANGED`	`DATA_CHANGE`
`DRAW`	`UPDATE_COMPLETE`
`VALUE_COMMITED`	`VALUE_COMMIT`

mx.events.IndexChangedEvent

The following table describes changes to the IndexChangedEvent class:

Member	Change description
`inputType`	`triggerEvent`

mx.events.ItemClickEvent

The following table describes changes to the ItemClickEvent class:

Member	Change description
`data`	`item`
`relatedNode`	`relatedObject` and has a type of InteractiveObject (not DisplayObject)

mx.events.ListEvent

The following table describes changes to the ListEvent class:

Member	Change description
`cell`	`itemRenderer`
`CELL_BEGIN_EDIT`	`ITEM_EDIT_BEGIN`
`CELL_END_EDIT`	`ITEM_EDIT_END`
`CELL_FOCUS_IN`	`ITEM_FOCUS_IN`
`CELL_FOCUS_OUT`	`ITEM_FOCUS_OUT`
`CELL_REQUEST_EDIT`	`ITEM_EDIT_BEGINNING`
`cellRenderer`	`itemRenderer`
`itemIndex`	`rowIndex`
`itemSkin`	`itemRenderer`

mx.events.ListItemSelectEvent

The following table describes changes to the ListItemSelectEvent class:

Member	Change description
`cellRenderer`	`itemRenderer`
`inputType`	`triggerEvent`
`itemSkin`	`itemRenderer`

mx.events.LowLevelEvents

This class has been removed.

mx.events.MenuEvent

The following table describes changes to the MenuEvent class:

Member	Change description
`cellRenderer`	`itemRenderer`
`change`	`itemClick`
`itemSkin`	`itemRenderer`
`menuItem`	`item`

Member	Change description
`menuItemRollOut`	`itemRollOut`
`menuItemRollOver`	`itemRollOver`

mx.events.MouseEvent

The MouseEvent class is now called FlexMouseEvent. Do not confuse this with the flash.events.MouseEvent class, which still exists.

The following table describes changes to the mx.events.MouseEvent class:

Member	Change description
`MOUSE_SCROLL_OUTSIDE`	`MOUSE_WHEEL_OUTSIDE`

mx.utils.events.ObjectEvent

The ObjectEvent class is now named and is moved to mx.events.PropertyChangeEvent.

mx.utils.events.ObjectEventKind

The ObjectEventKind class is now named and is moved to mx.events.PropertyChangeEventKind.

mx.events.SliderEvent

The following table describes changes to the SliderEvent class:

Member	Change description
`inputType`	`triggerEvent`
`newValue`	`value`

mx.events.ToolTipEvent

The following table describes changes to the ToolTipEvent class:

Member	Change description
`CREATE_TOOL_TIP`	`TOOL_TIP_CREATE`
`END_TOOL_TIP`	`TOOL_TIP_END`
`HIDE_TOOL_TIP`	`TOOL_TIP_HIDE`

Member	Change description
SHOW_TOOL_TIP	TOOL_TIP_SHOW
SHOWN_TOOL_TIP	TOOL_TIP_SHOWN
START_TOOL_TIP	TOOL_TIP_START

mx.events.TreeEvent

The following table describes changes to the TreeEvent class:

Member	Change description
CELL_PRESS	Removed.
cellRenderer	itemRenderer
inputType	triggerEvent
itemSkin	itemRenderer

mx.events.UIEventDispatcher

This class has been removed.

Formatters

This section describes changes to formatters.

mx.formatters.DateFormater

The DateFormatter pattern string can contain other text in addition to pattern letters. In Flex 1.5, the pattern string had to end with a pattern letter, and text following the last pattern letter was truncated.

In Flex 2.0, you can have additional text after the last pattern letter and that text is no longer truncated.

mx.formatters.NumberBase

Four public properties of the mx.formatters.NumberBase class have been renamed to be consistent with similar mx.formatters and mx.validators properties.

The following table describes the changes to the NumberBase class:

Member	Change description
`dSymbolFrom`	`decimalSeparatorFrom`
`dSymbolTo`	`decimalSymbolTo`
`tSymbolFrom`	`thousandsSeparatorFrom`
`tSymbolTo`	`thousandsSeparatorTo`

mx.formatters.SwitchSymbolFormatter

The following table describes the changes to the SwitchSymbolFormatter class:

Member	Change description
`isValid`	This is now private.
`numberSymbol`	This is now private.

Interfaces

The following table describes changes to Flex interfaces:

Old name	New name
IAllChildrenContainer	mx.core.IRawChildrenContainer
IAllChildrenList	Removed. Refer to SystemManager.numChildren and SystemManager.getChildAt.
IChildrenList (formerly IChildCollection)	mx.core.IChildList
IContainer	Added as a marker interface to replace IFlexContainer in the compiler.
ICreatedComponents	Removed. Refer to mx.core.Container instead.
IDataObject	mx.core.IDataRenderer
IDeferredInstantiationContainer	Removed. Refer to mx.core.Container instead.
IDeferredInstantiationUIComponent. cacheHeuristic	The `cacheHeuristic` property has been made private.
IFlexContainer	Removed. The framework code refers to mx.core.Container instead. The compiler refers to IContainer instead.
IFocusable	mx.managers.IFocusManagerComponent

Old name	New name
IFocusManager	Removed. Refer to mx.managers.FocusManager instead.
IFocusManagerContainer	mx.managers.IFocusManagerContainer
IHistoryState	mx.managers.IHistoryManagerClient
IInteractionReplayer	mx.automation.IAutomationReplayer
ILayoutClient	mx.managers.ILayoutManagerClient
ILayoutManager	mx.managers.LayoutManager
IObjectChanged	mx.core.IPropertyChangeNotifier
IRecorder	Removed.
IRepeaterContainer	Removed. Refer to mx.core.Container instead.
IScrollBar	Removed. Refer to mx.controls.ScrollBar instead.
IStyleable	mx.styles.ISimpleStyleClient
IStyleClient	mx.styles.IStyleClient
ISystemManager	The framework refers to mx.managers.SystemManager instead. References to ISystemManager have been removed from the compiler.
ITabGroup	mx.managers.IFocusManagerGroup
ITabularData	mx.automation.IAutomationTabularData
ITarget	mx.logging.ILoggingTarget
IToolTipClient	mx.managers.IToolTipManagerClient
ITreeDataProvider	Removed.
IValidationListener	mx.validators.IValidatorListener

Managers

This section describes changes to Flex classes in the mx.managers package.

mx.managers.DepthManager

This class has been removed.

mx.managers.DragManager

Instead of setting event.handled inside the dragEnter event listener, you must call the `DragManager.acceptDragDrop(event.target)` method.

You cannot set the `action` property of the event object. Instead, you must call the `setFeedback()` method. For example, change the following code:

```
event.action=DragManager.Link;
```

to the following:

```
DragManager.showFeedback(DragManager.LINK);
```

This applies to all actions: `MOVE`, `LINK`, `COPY`, `NONE`.

The signature to the `doDrag()` method has changed. There is no longer an `imageInitObj` argument, and there are two new arguments: `mouseEvent` and `allowMove`.

When using the `doDrag()` method and dragging text-based controls, you should use the `currentTarget` property of the Event object instead of the `target` property. This is because of the way text-based controls react during the bubbling event phase.

For a detailed description of these changes and a conversion example, see "Using the drag-and-drop feature" on page 178.

mx.managers.FocusManager

The following table describes changes to the FocusManager class:

Member	Change description
`isParentOf()`	Removed. Use the `contains()` method on the DisplayObjectContainer class.

mx.managers.LayoutManager

The following table describes changes to the LayoutManager class:

Member	Change description
`updateNow()`	`validateNow()`

mx.managers.PopUpManager

The following table describes changes to the PopUpManager class:

Member	Change description
`closeButton`	The default value of the `closeButton` property is now `false`. To enable a close button on your pop-ups, you must explicitly set the value of the `closeButton` property to `true`.
`createPopUp()`	The `createPopUp()` method now takes only three arguments, and returns an IFlexDisplayObject. You can no longer pass an initObj to the `createPopUp()` method. Instead, you declare the variables that you want to set inside the pop-up's definition, and set them on the pop-up in the calling application.
`deletePopUp()`	`removePopUp()`
`popUpWindow() (was popupWindow())`	The name of the method also changed to `addPopUp()`. Formerly inherited from the Application object, this method is now a method of the PopUpManager class.

mx.managers.SystemManager

The SystemManager class now implements the ISystemManager interface.

The following table describes changes to the SystemManager class:

Member	Change description
`cursors` (formerly `cursorChildrenList`)	`cursorChildren`
`embeddedFontList()`	This method is now private. Use the Font class's `enumerateFonts()` method.
`getClassByName()`	`getDefinitionByName()`
`getManager()`	`getSystemManager()`
`getTopLevelSystemManager()`	`topLevelSystemManager`
`initializeChild()`	`childAdded()`
`registerInitCallback()`	This method is now private.
`toolTips` (formerly `toolTipChildrenList`)	`toolTipChildren`
`topMostChildrenList`	`popupChildren`

Media controls

This section describes changes to media controls in Flex 2.

mx.controls.MediaDisplay

The MediaDisplay control has been replaced by the mx.controls.VideoDisplay control. The API is the same, but it does not support MP3 files.

mx.controls.MediaController

The MediaController control has been removed. Use the VideoDisplay control instead.

mx.controls.MediaPlayback

The MediaPlayback control has been removed. Use the VideoDisplay control instead.

mx.controls.VideoDisplay

This class replaces the mx.controls.MediaDisplay class. The following table describes changes to the VideoDisplay class:

Member	Change description
`metadataReceived`	Removed.

Printing

This section provides information about changes to classes in the mx.printing package.

The mx.print package is now named mx.printing.

mx.print.PrintJob

The name and package of the mx.print.PrintJob class has been changed to mx.printing.FlexPrintJob.

mx.print.PrintJobType

The following table describes the changes to the PrintJobType class:

Member	Change description
`HEIGHT`	`MATCH_HEIGHT`
`WIDTH`	`MATCH_WIDTH`

Resources

This section provides information about changes to classes in the mx.resources package.

mx.resource.*

The mx.resource package is now named mx.resources.

Service tags

This section describes changes to Flex classes in the mx.servicetags package. In addition to the changes here, the Flex data services architecture has changed. For more information, see Chapter 9, "Data Services," on page 135.

mx.servicetags.HTTPService

The HTTPService class was moved to mx.rpc.http.HTTPService. It now extends the mx.rpc.AbstractInvoker class.

The following table describes changes to the HTTPService class:

Member	Change description
`protocol`	Deprecated. Use the `destination` property. For backward compatibility, the default value of the `destination` property is `defaultHttps`.
`serviceName`	Deprecated. Use the `destination` property.

The HTTPService logic has been changed to match the WebService logic. Previously, HTTPServices raised a fault for an invalid `resultFormat` at send time; now, HTTPService throws an Error as soon as an invalid value is set.

mx.servicetags.RemoteObject

The RemoteObject class was moved to mx.rpc.remoting.RemoteObject. It now extends the mx.rpc.AbstractService class.

The following table describes changes to the RemoteObject class:

Member	Change description
`encoding`	Removed.
`endpoint`	Was removed. Channels defined in the flex-services.xml file replace the need for this property. Has been added again to allow clients to use this tag without the configuration file at compile time.
`named`	Removed. Use the `destination` property.
`protocol`	Removed. Channels defined in the flex-services.xml file replace the need for this property.
`source`	Removed. Use the `destination` property.
`type`	Removed. This is now controlled in the service definition in the flex-services.xml file. You set it with the `<source>` tag.

All RemoteObject sources need destination entries in the flex-services.xml file. You should now name every service and not use the actual source class as the name. All definitions of named RemoteObjects must be moved into the flex-services.xml file.

mx.servicetags.WebService

The following table describes changes to the WebService class:

Member	Change description
`protocol`	Deprecated. Use the `destination` property. For backward compatibility, the default value of the `destination` property is `defaultHttps`.
`serviceName`	Deprecated. Use the `destination` property.

The WebService class has the following changes:

- The WebService class was moved to mx.rpc.soap.WebService. It now extends the mx.rpc.AbstractService class.
- You must move the definitions of named services to the flex-services.xml file. The unnamed whitelist must be updated in the defaultHttp destination.

Callback URLs

Callback URLs are no longer necessary in Flex 2. Use messaging channels with endpoint mappings instead.

Skins

This section describes changes to Flex classes in the mx.skins package. For more information, see "Using skinning" on page 124.

mx.skins.halo.RectBorder

This class is now named HaloBorder.

mx.skins.halo.PopUpIcon

The following table describes changes to the PopUpIcon class:

Member	Change description
`arrowColor`	This property is now private.

mx.skins.ProgrammaticSkin

The following table describes changes to the ProgrammaticSkin class:

Member	Change description
`invalidateStyle()`	`styleChanged()`
`updateNow()`	`validateNow()`

mx.skins.RectBorder

This class is now named mx.skins.RectangularBorder.

States

This section provides information about changes to classes in the mx.states package.

mx.states.AddChild

The following table describes changes to AddChild class:

Member	Change description
`added`	This property is now internal.
`instanceCreated`	This property is now internal.
`target`	`relativeTo`

mx.states.SetEventHandler

The following table describes changes to SetEventHandler class:

Member	Change description
`event`	`name`

mx.states.SetProperty

The following table describes changes to SetProperty class:

Member	Change description
`property`	`name`

mx.states.SetStyle

The following table describes changes to SetStyle class:

Member	Change description
`property`	`name`

Utilities

This section provides information about changes to classes in the mx.utils package.

mx.utils.XMLUtil

The following table describes changes to XMLUtil class:

Member	Change description
`createXML()`	`createXMLDocument()`

Validators

This section provides information about changes to classes in the mx.validator package.

mx.validator.Validator

The following table describes the changes to the Validator class:

Member	Change description
`DIGITS`	`DECIMAL_DIGITS`
`disable()`	`enabled`
`disableStructure`	`enabled`
`enable()`	`enabled`
`enableStructure()`	`enabled`
`field`	`source` and `property`
`hasErrors()`	Removed.
`isStructureValid()`	`validate()`
`isValid()`	`validate()`
`LETTERS`	This property was private and is now protected; it is now named `ROMAN_LETTERS`.
`validateAll()`	Returns an Array of ValidationResultEvent for Validators that failed. If all are successful, this method returns an empty Array. Previously, this method returned a Boolean value.
`validationError()`	Removed.

CHAPTER 4

Data Providers

4

This topic describes migrating data providers, including the `dataProvider` property of Flex controls and the ways you access and manipulate the data represented by the `dataProvider` property. For detailed information on using data providers in Flex 2, see Chapter 7, "Using Data Providers and Collections," in *Flex 2 Developer's Guide*.

The DataProvider interface and class no longer exist and have been replaced by the collection package hierarchy. However, you still use the `dataProvider` property to specify the source of the data in a control such as DataGrid or Menu. The collection classes include methods for manipulating the underlying data and the view of that data that is displayed in the control.

The collection package includes the following interfaces and classes:

- IList, ICollectionView, and IViewCursor interface, and the CursorBookmark class which you use to access and manipulate data. The ICollectionView interface can represent a sorted or filtered subset of data without modifying the underlying data.
- ArrayCollection class, which implements the IList and ICollectionView interface using a backing Array.
- XMLListCollection which implements the IList and ICollectionView interface using a backing E4X XML object.
- Sort, and SortField for sorting the data representation of an ICollectionView.
- ItemResponder, for handling remote paged collections.
- ListCollectionView, a building-block class used by XMLListCollection and ArrayCollection.

For detailed descriptions of all interfaces and classes, see the collection package in *Adobe Flex 2 Language Reference*. For documentation on using the package interfaces and classes, see Chapter 7, "Using Data Providers and Collections," in *Flex 2 Developer's Guide*.

The following information briefly describes major migration issues:

- Because the DataProvider interface has been replaced by methods of the collection classes, controls that have dataProvider properties do not include any of the DataProvider interface methods, such as getItemAt(). Instead, you manipulate the control contents by manipulating the object, normally a collection, that acts as the data source. You can use the methods on the `dataProvider` property or on the collection object directly.

  For example, you can no longer use the following line:

  ```
  myList.getItemAt(index).
  ```

  Instead, use the following line:

  ```
  myList.dataProvider.getItemAt(index)
  ```

  In this case, the dataProvider property must specify a collection object that represents the data.

- Do not use raw Arrays or Objects in your dataProvider property if the data provider's values change. The control that displays the data will not get updated when the underlying data changes. Instead, convert your provider to an ArrayCollection class, as in the following example:

  ```
  <?xml version="1.0"?>
  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
    <mx:Script>
      import mx.collections.*;
      public var myArray:Array = ["MA", "ME", "MI", "MN","MO", "MS",
    "MT"];
      public var myICV:ICollectionView = new ArrayCollection(myArray);
    </mx:Script>
    <mx:ComboBox id="myCBO" dataProvider="{myICV}" />
  </mx:Application>
  ```

- If you use the DataProvider class directly in an existing application, replace it with the ArrayCollection or XMLListCollection class. These classes are not one-for-one equivalents to the old DataProvider class. They do not include the editField, getEditingData, getItemID, replaceItemAt, sortItems and sortItemsBy methods. If you use any of these methods, you must different techniques. For sorting, use the Sort class and the sort property of the ICollectionView interface.

- The Flex 1.5 Array class implemented the DataProvider interface methods, such as replaceItemAt() and sortItems(). In Flex 2, there is no separate Flex Array class, only the Flash Array class, which does not implement such methods. If you used the Array class and its DataProvider methods, convert the Array to an ArrayCollection and use IList and ICollectionView interface methods.

- The Tree, Menu, MenuBar, and PopUpMenuButton controls, now use a data descriptor class to access and manipulate control-specific information, such as menu item type, that is contained in the data provider. Tree controls require a class that implements the ITreeDataDescriptor interface, and menu-based controls require a class that implements the IMenuDataDescriptor interface. The Tree and menu-based controls use a DefaultDataDescriptor class, which implements both interfaces, as the data descriptor class unless you specify a custom data descriptor.

  The DefaultDataDescriptor supports standard E4X XML data sources, and Array based-sources that conform to specific structure rules. You might have to change the structure of your Array or object-based data source to conform to data descriptor rules. For detailed information see “Using hierarchical data providers” in Chapter 4, “Data Providers,” in *Flex 2 Developer’s Guide*.
- To sort the contents of a DataGrid control, you now sort the underlying data provider, using the collection interfaces. You can still let users control sorting by clicking on the grid headers. For detailed information see “Sorting data in DataGrid controls” in Chapter 12, “Using Data-Driven Controls,” in *Flex 2 Developer’s Guide*

CHAPTER 5

Binding

5

This topic describes the steps required to convert properties in your applications to be usable as the source for a data binding expression in Adobe Flex.

For more information binding, see Chapter 5, “Using Metadata Tags in Custom Components,” in *Creating and Extending Flex 2 Components*.

Contents

About binding

In Flex 1.5, it was possible to use any public property defined as a variable or Array as the source for a data binding expression. When a public property is the source of a data binding expression, Flex automatically copies its value to a destination property when the value of the source property changes.

Any property can be the destination of a data binding expression. That means you do not have to write an special code to support the destination property of a data binding expression.

Using any property as the source for data binding is supported in Flex 2, but you must explicitly identify the property or the property’s class as bindable using the `[Bindable]` metadata tag. The property may be defined as a variable or defined by using both a setter and a getter method. If you attempt to use a property as the source of a data binding expression, and that property does not support binding, Flex throws a warning and does not update the destination value of the when the source value changes.

Using Arrays and DataProviders as the source for data binding expressions in Flex 2 is not supported and you should convert your DataProviders to use the new Collections API. For more information, see “Binding with Arrays” on page 110.

The <mx:Binding> tag must be a top-level tag

In Flex 1.5, you could place the `<mx:Binding>` tag in a Flex container. In Flex 2.0, the `<mx:Binding>` tag must be a top-level tag in the MXML file. For example:

Flex 1.x:

```
<mx:HBox>
  <mx:Label id="myLabel"/>
  <mx:Label id="my2ndLabel" text="hello"/>
  <mx:Binding source="my2ndLabel.text" destination="myLabel.text"/>
</mx:HBox>
```

Flex 2:

```
<mx:Binding source="my2ndLabel.text" destination="myLabel.text"/>
<mx:HBox>
  <mx:Label id="myLabel"/>
  <mx:Label id="my2ndLabel" text="hello"/>
</mx:HBox>
```

Binding from a property

To make properties usable as the source for a data binding expression in Flex 2, you use the `[Bindable]` metadata tag. Properties of custom components often have getter/setter pairs that were tagged with a `[ChangeEvent]` metadata tag. For Flex 2, you convert that tag to `[Bindable]` and update the setter to dispatch an Event object.

This section describes these processes.

Binding from all public properties in a class

To make all public properties in a class usable as the source for a data binding expression, properties defined as properties and properties defined by using both a setter and a getter method, add the `[Bindable]` metadata tag before the class statement:

```
[Bindable]
public class MyClass { ... }
```

Although the easiest way to migrate an application that uses binding is to make all public properties in a class support data binding, it is not necessarily the best practice to use the `[Bindable]` metadata tag on an entire class. Doing this causes the compiler to generate more code, which in turn can affect performance and increase your application's file size.

The Flex compiler automatically generates an event named `propertyChange` for all public properties so the properties can be used as the source of a data binding expression. In this case, specifying the `[Bindable]` metadata tag with no event is the same as specifying the following:

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

Binding from a single property

To make a single property usable as the source for a data binding expression, add the `[Bindable]` metadata tag before the property declaration. The property can be public, protected, or private.

Flex 1.5:

```
var foo:String; // You could bind to this in 1.5
```

Flex 2:

```
[Bindable]
public var foo:String;
```

The Flex compiler automatically generates an event named `propertyChange` for all public properties so the properties can be used as the source of a data binding expression. In this case, specifying the `[Bindable]` metadata tag with no event is the same as specifying the following:

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

Binding from a property defined by a setter and getter method

In Flex 1.5, you used the `[ChangeEvent]` metadata tag on setter/getter pairs. In Flex 2, you replace the `[ChangeEvent]` metadata tag with the `[Bindable]` metadata tag and dispatch the event in the method. This strategy also applies to other methods that dispatched custom events.

Flex 1.5:

```
[ChangeEvent("maxFontSizeChanged")]
// Define public getter method.
public function get maxFontSize():Number {
  return _maxFontSize;
}
```

Flex 2:

```
[Bindable(event="maxFontSizeChanged")]
// Define public getter method.
public function get maxFontSize():Number {
  return _maxFontSize;
}
```

The following is a Flex 2 example of a getter/setter pair that uses the `[Bindable]` metadata tag:

```
// Define private variable.
private var _maxFontSize:Number = 15;

[Bindable(event="maxFontSizeChanged")]
// 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;

  // Create event object.
  var eventObj:Event = new Event("maxFontSizeChanged");
  dispatchEvent(eventObj);
}
```

You can omit the event name in the metadata tag, as the following example shows:

```
  // Define private variable.
  private var _maxFontSize:Number = 15;

  [Bindable]
  // 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;
  }
```

The Flex compiler automatically generates an event named `propertyChange`. In this case, specifying the `[Bindable]` metadata tag with no event is the same as specifying the following:

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

Dispatching binding events from a custom component

To dispatch an event to trigger data binding for a property, the property's class must either extend EventDispatcher or implement the IEventDispatcher interface. Be sure to update the object that is passed to the `dispatchEvent()` method. You must use the new Event class and not a generic Object. For more information on converting events, see "Migrating the Event object" on page 113.

If the class already extends EventDispatcher, you do not need to make any changes. If the class already implements IEventDispatcher, it must implement `dispatchEvent()`, too. If the class does not implement `dispatchEvent()`, the MXML compiler reports a warning.

If the class does not extend EventDispatcher or implement IEventDispatcher and one of your properties is marked `[Bindable]` or the class is marked `[Bindable]`, the MXML compiler modifies the class to implement IEventDispatcher. This requires the compiler to generate the following code for you:

- Implements the IEventDispatcher interface.
- Adds an `addEventListener()` method.
- Adds a `removeEventListener()` method.
- Adds a `dispatchEvent()` method.

For example, the following declaration:

```
class Foo {
}
```

Is converted by the Flex compiler to the following:

```
class Foo implements IEventDispatcher {
   private var bar:EventDispatcher = new EventDispatcher(this);
   public function addEventListener(type:String, listener:Object,
     useCapture:Boolean = false, priority:int = 0):Boolean {
     return bar.addEventListener(type, listener, useCapture, priority);
   }
   public function removeEventListener(type:String, listener:Object,
     useCapture:Boolean = false):Boolean {
     return bar.removeEventListener(type, listener, useCapture);
   }
   public function dispatchEvent(event:Event):void {
     bar.dispatchEvent(event);
   }
}
```

For more information, see *Creating and Extending Flex 2 Components*.

Binding with Flex component properties

You can no longer use all properties of Flex components as the source of a data binding expression without extending those controls. Properties of Flex components that can be used as the source of a data binding expression contain the following description in their entry in the *ActionScript 3.0 Language Reference*:

```
This property can be used as the source for data binding.
```

Binding with Arrays

In Flex 1.5, the DataProvider class had convenience functions that it inherited from Array such as `addItem()` and `removeItemAt()`. These functions dispatched events so that Arrays and Array subclasses could be used as the source of a data binding expression. In Flex 2, Arrays supports only one-time binding. If you want your DataProviders to support binding, you must convert them to Collections.

For more information on migrating applications that use the DataProvider class, see "mx.controls.listclasses.DataProvider" on page 70.

For more information on using Collections, see *Flex 2 Developer's Guide*.

CHAPTER 6

Events

6

This topic describes changes to the Adobe Flex 2 event model for developers who are migrating Flex applications.

Contents

About events

The following list is a general overview of the changes to the Event model. Review all event handling in your Flex application by using these guidelines.

- All event objects are either of type Event or a subclass of Event. You should explicitly declare or cast them to their appropriate type.
- Use static constants such as `MouseEvent.CLICK` instead of string literals for the event type, such as "`click`". For more information, see "Using static constants" on page 114.
- Do not use object event listeners. Instead use function listeners. For more information, see "Using function listeners" on page 115.
- Scoping in event listeners is improved. You no longer need to pass a Delegate to maintain scope.
- Use the `currentTarget` property instead of the `target` property when you access the object that is listening for the event. The `target` property refers to the object that dispatched the event.

Subclasses of Event, such as MouseEvent, have all the properties of the Event object and properties that are specific to that type of event. Choosing the most specific type possible provides the following benefits:

- Faster run-time performance
- Compile-time type-checking
- Access to event-specific properties
- Smaller SWF file size

Component startup life cycle

The ordering of events during a component's initialization has changed. The `initialize` event is now named `preinitialize`.

The `initialize` event now occurs after children are added to the component. As a result, instead of calling an event handler for the `creationComplete` event, you can use the `initialize` event handler to perform most initialization tasks. You typically call `creationComplete` when you have to wait until the LayoutManager has processed the children so that the `x`, `y`, `width`, and `height` properties are known.

The `childrenCreated` event has been removed. You should use the `initialize` event instead.

A new event, `applicationComplete`, has been added. This event is the last one dispatched when an application starts up.

Migrating the Event object

This section describes migration issues related to accessing the Event object.

Using the Event object

The Event object is no longer of type Object. It is now of type flash.events.Event. Specify a stricter type in functions, as follows:

```
private function eventHandler(event:Object):Void { // Flex 1.5
```

```
private function eventHandler(event:Event):void { // Flex 2
```

You should also now cast the `event.target` in an event handler to an appropriate type to avoid compile-time warnings. The new code should specify the stricter event type as a function parameter, as the following example shows:

```
private function doDragExit(event:DragEvent):void {
   event.target.hideDropFeedback(event);
}
```

If you assign the target to a stronger-typed variable, you must cast; for example:

```
var clickTarget:Button = Button(event.target);
```

When using the Event object, you are encouraged to use the most strict type possible. This lets you access properties that are specific to the target type. For example, in a mouse click listener, you should declare the Event of type MouseEvent rather than of type Event, as follows:

```
private function myClickHandler(event:MouseEvent):void { ... }
```

If you use Event as the type, rather than MouseEvent, you do not have access to any properties that are specific to the MouseEvent class.

Using the target property

Calling methods and accessing properties on the target can be confusing. The default type of Event.target is Object. Because ActionScript is strongly typed, you can call `event.target.`*`methodName`*`()` only if you cast the `event.target` to an object type that defines that *`methodName`*. The same applies for properties. You can access `event.target.`*`property`* only if you define *`property`* on the new type.

If you try to call another method on the target (for example, the `getStyle()` method), Flex returns an error. The `getStyle()` method is a method of UIComponent, a subclass of DisplayObject. Therefore, you must cast event.target to UIComponent before calling the `getStyle()` method, as the following example shows:

```
function myEventListener(e:Event) {
   UIComponent(e.target).getStyle("color");
}
```

If you use the `target` property of the Event object to determine which control triggered the event, you should consider changing the `target` property to `currentTarget`. The `currentTarget` property supports event bubbling and capturing, which are new in Flex 2. During these phases, the event is actually being handled by the target's parent containers. The target still refers to the dispatcher of the event, but the `currentTarget` refers to whatever control is currently processing a bubbling or capture event.

For example, if you have `<mx:List click="..."/>`, inside the click handler `event.target` might be one of the rows, but `event.currentTarget` is a reference to the List control, which is generally what you would expect.

Another example is what occurs when you assign a mouseDown handler on a TextInput control. Actually, the TextField class that is inside the TextInput control dispatches the event (depending on where the user clicked), so the TextField control, not the TextInput control, is the `target`. However, the TextInput control is the `currentTarget` if that is where you attached the handler.

Most controls have internal subcomponents that are often the target for mouse events.

Using static constants

Use static constants to represent event types; for example, use `MouseEvent.CLICK` rather than `"click"`, as the following example shows:

```
addEventListener("click", myClickListener); // Flex 1.5
addEventListener(MouseEvent.CLICK, myClickListener); // Flex 2
```

Change the following:

```
switch (event.type) {
  case "click":
  ...
}
```

To this:

```
switch (event.type) {
  case MouseEvent.CLICK:
  ...
}
```

To find the appropriate static constant for your event type, see the events section of the control's entry in *Adobe Flex 2 Language Reference*.

Using function listeners

When migrating, convert all object event listeners to function event listeners. You can no longer pass an object as the second parameter to the `addEventListener()` method. The listener argument of the `addEventListener()` method was of type Object, which also accepted a Function, but is now of type Function. If you try to pass an object listener, Flex reports an error.

For example, if you had the following:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  creationComplete="createHandler()">
   <mx:Script>
   import mx.core.Alert;
   function createHandler() {
     var myListener = new Object();
     myListener.click = function(event) {
       Alert.show("This is a log message");
     }
     myButton.addEventListener("click", myListener);
     trace("Added listener");
   }
   </mx:Script>
   <mx:Button label="Click Me" id="myButton"/>
</mx:Application>
```

Convert it to the following:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="createHandler()">
   <mx:Script>
   import mx.controls.Alert;
   private function createHandler():void {
     trace("Added listener");
     button1.addEventListener(MouseEvent.CLICK, myClickHandler);
   }
   private function myClickHandler(event:MouseEvent):void {
     Alert.show("This is a log message");
   }
   </mx:Script>
   <mx:Button label="Click Me" id="button1"/>
</mx:Application>
```

You must also change your code if you created a custom event handler class and defined a `handleEvent()` method that listened for all events. Flex implicitly registered this method as a handler for all events. Functions named `handleEvent()` no longer catch all events by default as they did in Flex 1.x.

For example, if in your application you registered the custom listener with the `addEventListener()` method, as follows:

```
public var myListener:MyEventListener = new MyEventListener();
b1.addEventListener("click", myListener);
```

You now explicitly register the custom listener's `handleEvent()` method for each event you want to handle, as follows:

```
public var myListener:MyEventListener = new MyEventListener();
b1.addEventListener(MouseEvent.CLICK, myListener.handleEvent);
```

Using the EventDispatcher class

The UIComponent class now inherits from the player's EventDispatcher class. As a result, you no longer need to mix in the EventDispatcher class when creating new components.

The `dispatchEvent()` method now requires an argument of type Event, so you can no longer construct an event using an Object like `{ type: "click" }`. You must now do the following instead:

```
dispatchEvent(new MouseEvent(MouseEvent.CLICK));
```

To dispatch an event from a custom component, you can do the following:

Flex 1.x:

```
<mx:CustomPanel mouseDown="dispatchEvent({type: 'resizeEvent', size:
  'small'})">
```

Flex 2:

```
private function mouseDownHandler(event:MouseEvent):void {
  var resizePanelEvent:ResizePanelEvent = new
    ResizePanelEvent(ResizePanelEvent.RESIZE);
  resizePanelEvent resizePanelEvent.size = "small";
  dispatchEvent(event);
}
```

Maintaining scope

Previously, you used the mx.utils.Delegate class to provide access to the document scope within an event handler. You would pass the Delegate object into a call to the `addEventListener()` method so that the listener would execute in that scope. This is no longer necessary. The event listener's scope is now the class in which it is declared as a function, which is in most cases the scope you should expect; for example:

Flex 1.x:

```
addEventListener("click", mx.utils.Delegate.create(this, myListener));
```

Flex 2:

```
addEventListener(MouseEvent.CLICK, myListener);
```

Using setCapture()

The `setCapture()` method has been removed. This method was added in earlier Beta releases of Flex 2 and was used to block events going to other (nested or non-nested) components during drag-and-drop operations and a few other interactions. However, in the nested situation, if one component called the `setCapture()` method, its parents and children could not.

If you require this functionality, you should use the capture phase of the event model by calling the `stage.addEventListener("mouseXXX", mouseXXXHandler, true)` method, where `mouseXXX` is the mouse event you want to block. You can then emulate the `setCapture()` method by calling the `event.stopPropagation()` method; however, Adobe does not recommend doing this because most components should ignore the event.

The `removeCapture()` method has also been removed.

Keyboard events

The `code` and `ascii` properties of the KeyboardEvent (formerly Event) object are now `keyCode` and `charCode`.

In drag-and-drop examples in Flex 1.x, you could use `Key.isDown()` to detect a key that was pressed during the operation. In Flex 2, you use the `shiftKey`, `ctrlKey`, and `altKey` properties on the KeyboardEvent object to detect keys that are pressed during the operation.

CHAPTER 7

Styles and Skinning

7

This topic describes styleable objects; new skinning workflow; and CSS rules. For information about changes to individual classes such as ProgrammaticSkin, see Chapter 3, “Flex Classes,” on page 41.

Contents

Using styles

There are some changes to the way you use styles. For example, the use of style properties is now more strictly enforced. If you try to apply a style to a component that does not support that style, Flex throws and error. Also, you cannot use the `StyleManager.styles` property to change styles for entire classes in Adobe Flex Software Development Kit (SDK) 2. Finally, some of the color value formats have changed.

This section describes these changes.

Using the StyleManager

When you use the StyleManager to apply styles to entire classes, you must now access the class with the `getStyleDeclaration()` method. For example:

Flex 1.x:

```
StyleManager.styles.Button.setStyle("color","red");
```

Flex 2:

```
StyleManager.getStyleDeclaration("Button").setStyle("color","red");
```

You can no longer set or get styles as properties of a component or class. You now use `setStyle()` and `getStyle()`. This also applies to using StyleManager. While accessing styles this way was discouraged in Flex 1.5, it was not prohibited until now.

Flex 1.x:

```
var c = myButton.color;
StyleManager.styles.TextArea.color = "red";
```

Flex 2:

```
var c:Number = myButton.getStyle("color");
StyleManager.getStyleDeclaration("TextArea").setStyle("color","red");
```

In addition, you can no longer create a CSSStyleDeclaration object and then apply it to a type of control, as the following example shows:

```
public var styleObj:CSSStyleDeclaration = new CSSStyleDeclaration();
styleObj.setStyle("color","red");
styleObj.setStyle("fontFamily","Tahoma");
StyleManager.styles.Accordion = styleObj;
```

You must now use the `setStyle()` method; for example:

```
StyleManager.getStyleDeclaration("Accordion").setStyle("color","red");
StyleManager.getStyleDeclaration("Accordion").setStyle("fontFamily","Tahoma
  ");
```

You can also use the `StyleManager.setStyleDeclaration()` method.

The `getNonInheritingStyle()` and `getInheritingStyle()` methods were removed. You can now use the `getStyle()` method instead.

Changed style properties

The following table shows changes to the CSS style property names:

Flex 1.x Name	Flex 2 Name
`drop-shadow`	`drop-shadow-enabled`
`margin-bottom`	`padding-bottom`
`margin-left`	`padding-left`
`margin-right`	`padding-right`
`margin-top`	`padding-top`
`tool-tip-offset`	`data-tip-offset`
`tool-tip-precision`	`data-tip-precision`
`tool-tip-placement`	`data-tip-placement`
`track-highlight`	`show-track-highlight`

Missing style properties

If you applied a style property to a component that did not support that style property, Flex would fail silently. Now, Flex enforces style setting and throws compilation errors when you try to apply a style to a component that does not support that style. You should check the component's entry in *Adobe Flex 2 Language Reference* to see the legal styles for each component you use.

In Flex 1.5:

```
<mx:Button label="my link" borderStyle="solid"/>
```

In Flex 2, the compiler throws an error for the `borderStyle` style because it's not a valid style on a Button control.

Using colors

The support color value formats have changed. Which format is supported depends on whether you are setting the color in CSS or using the `setStyle()` method. This section describes the valid and invalid formats.

Color value formats in CSS

In CSS, color values must be un-quoted, and can be in either the `#RRGGBB` format or a constant (for example, `red`). These rules apply to CSS whether you are using the `<mx:Style>` tag or an external CSS file. Previously, you could use the `0xRRGGBB` notation or quoted values.

The following are the valid color value formats in CSS:

```
.b1 { color: red; }
.b2 { color: #66CC66; }
.b3 { color: rgb(22%,22%,77%); }
.b4 { color: rgb(255,0,0); }
.b5 { color: "0xFFFF00"; }
```

The following are the invalid color value formats in CSS:

```
.b6 { color: 0xFFFF00; } /* Results in a compiler error. */
.b7 { color: "red"; } /* Fails silently; color is black. */
.b8 { color: "#FFFF00"; } /* Fails silently; color is black. */
.b9 { color: 100%, 0%, 0% ; } /* Results in a compiler error. */
```

If the format does not conform to CSS syntax, Flex throws a compiler error and the application will not run. If the format does conform to CSS syntax but is uses invalid values, Flex throws a run-time error.

Color value formats using the StyleManager and setStyle() method

The validity of color formats is different when using the `setStyle()` method or when setting colors with the StyleManager.

The following are valid color formats when you use the `setStyle()` method or the StyleManager:

```
b1.setStyle("color",0x33CCFF);
b2.setStyle("color","0x33CCFF");
b3.setStyle("color","red");
b4.setStyle("color","#66CC66");
```

The following are invalid color formats when you use the `setStyle()` method or the StyleManager:

```
b5.setStyle("color",red);
b6.setStyle("color",rgb(0,255,0));
b7.setStyle("color",rgb(77%,22%,0%));
b8.setStyle("color",#66CC66);
```

CSS class selectors

The CSS syntax for class selectors is more strict. Class selectors now require a period as a prefix to the class selector name. In Flex 1.5, if a selector did not have a period prefixing its name, Flex interpreted it as a type selector, but then allowed you to use it as a class selector at run time.

For example:

```
<mx:Style>
   // The following class selector fails in Flex 2 if
   // there is no associated class:
   myClass {
     color: red;
   }
   // The following class selector works fine in Flex 2:
   .myOtherClass {
     color: red;
   }
</mx:Style>
```

You must include the period when you use the `getStyleDeclaration()` method.

Global style sheet

The global style sheet, global.css, was removed. Flex now uses a default style sheet, defaults.css, which is in the frameworks.swc file.

If you are migrating an existing 1.5 application that depended on the global.css file, you can still use global.css, but you must compile it into a theme (importing global.css is no longer automatic). For more information on compiling themes, see Chapter 18, “Using Styles and Themes,” in the *Flex 2 Developer’s Guide*.

Using type selectors

Properties set in type selectors now obey object-oriented inheritence rules. If you define a type selector for a class, Flex applies all properties in that type selector to that class, as well as subclasses of that class. For example, VBox is a subclass of Box, which is a subclass of Container.

If you define the following type selectors, VBox controls inherit the values of the `fontSize` and `fontStyle` properties.

```
<mx:Style>
  Container { fontSize:14 }
  Box { fontStyle: bold }
  VBox { color:red }
</mx:Style>
```

Previously, properties set in type selectors that were parents of a class did not apply to that class, unless that class was a child object in the application’s display list.

Using units

Flex no longer supports using the plus (+) and minus (-) unit modifiers.

Flex no longer supports the em and ex unit types.

Using skinning

Previously, you defined graphical skins as symbols in a FLA file, exported the FLA file as a SWC file from the Flash IDE, and added it to your Flex source path. The symbols in the new SWC file replaced existing symbols in Flex component skins.

Now, all skins are specified as style properties that can be set through CSS or inline. Do not use the symbol substitution method of defining new skins.

To use a programmatic skin, add the class to your ActionScript source path when you compile. Then use the `ClassReference()` statement to reference that class in your application's CSS. For example, if you have MySkins/MyButtonSkin.as, you use the following syntax to reference the class:

```
Button {
  upSkin: ClassReference("MySkins.MyButtonSkin");
}
```

You use the `Embed()` statement to reference graphic skins in CSS.

Many style properties that referred to skins are now deprecated. For example, the `brokenImage` property is obsolete and was replaced by the `brokenImageSkin` style property. This style property is of type Class. Flex throws an error if you try to apply a style to a property that no longer exists. For a list of available skin style properties, see the control's entry in the *Adobe Flex 2 Language Reference*.

The drawing methods such as `moveTo()` that you would use in programmatic skinning are now accessed through the `graphics` property of the MovieClip object. To use them you must import the flash.display.* package.

Skinning assets

The assets that Flex includes for you to use as a basis for reskinning components are changed.

Flex 1.x included:	Flex 2 includes:
• Sample programmatic skins in the flex_install_dir/resources/themes/ programmatic directory.	• HaloClassic skins for users who want the original look and feel of Flex applications.
• Graphical skins in the pulseBlue.fla and pulseOrange.fla theme files in the *flex_install_dir*/resources/themes/graphic directory.	• Halo programmatic skins, which are in the mx.skins.halo package. •
	• Graphical Aeon theme files, which are located in the framework/themes directory.

Drawing programmatic skins

The drawing methods are moved to the Graphics class, which is accessible through the `graphics` property.

For example, in Flex 1.x:

```
function draw() {
  clear();
  moveTo(0,0);
  lineTo(10,10);
}
```

In Flex 2:

```
import flash.display.Graphics;
function updateDisplayList(...) {
  var g:Graphics = graphics;
  g.clear();
  g.moveTo(0,0);
  g.lineTo(10,10);
}
```

Using embedded fonts

You mjust use some differences in the syntax for embedded fonts. In addition, the default font manager is changed.

The default font manager is actually both the Batik font manager and the JRE font manager. You set them in the list of font managers in the flex-config.xml file. This is a reverse ordered precedence. The preferred Batik font manager doesn't handle all embedded font commands, so what it doesn't handle, it passes to the JRE font manager. In Flex 1.5, the default font manager was the JRE Font Manager.

The following example shows the default setting for fontmanagers in Flex 1.5 (note that Batik is commented out):

```
<fonts>
   <managers>
     <manager-class>macromedia.fonts.JREFontManager</manager-class>
     <!-- <manager-class>macromedia.fonts.BatikFontManager</manager-class>
     -->
   </managers>
</fonts>
```

The following example hows the default font manager in Flex 2:

```
<fonts>
   <managers>
     <manager-class>flash.fonts.JREFontManager</manager-class>
     <manager-class>flash.fonts.BatikFontManager</manager-class>
   </managers>
</fonts>
```

NOTE

The class names of the font managers are changed from macromedia.* to flash.*.

Syntactically, you must now specify the font face in the selector or you receive a warning similar to the following:

```
"An embedded font was found for family 'myFont' but it did not have the
  requested plain font face."
```

In Flex 1.5, you could use the following syntax to embed a bold italic font face:

```
@font-face {
   src: url("GOTHICBI.TTF");
   font-style: italic;
   font-weight: bold;
   font-family: myFont;
}
.myStyleBoldItalic {
   font-family: myFont;
}
```

In Flex 2, you still use the `@font-face` rule to embed the font:

```
@font-face {
   src:url("GOTHICBI.TTF");
   font-style: italic;
   font-weight: bold;
   font-family: myFont;
}
```

But you then add descriptors to the class or type selector to use the embedded font face:

```
.myStyleBoldItalic {
   font-family: myFont;
   font-weight: bold;
   font-style: italic;
}
```

Themes

You can no longer export SWC files from Flash IDE and use the symbols in that SWC as part of a theme file in Flex. You must instead use the compc utility to compile a theme SWC file from CSS files and graphics.

The original Halo skins are repackaged into a theme so that you can change the appearance of your Flex applications back to the Flex 1.x look and feel.

For more information on creating and using themes, see the *Flex 2 Developer's Guide*.

CHAPTER 8

Behaviors

8

This topic describes the new architecture for behaviors in Macromedia Flex and syntax changes from Flex 1.5. For detailed information, see Chapter 17, “Using Behaviors,” in the *Flex 2 Developer’s Guide* and Chapter 15, “Creating Effects,” in *Creating and Extending Flex Components*.

Contents

Overview

Flex implements effects using an architecture in which each effect is represented by two classes:

- **Factory class** Creates an object of the instance class to perform the effect on the target. You create instances of the factory class 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 a 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** 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 an instance class is the name of the effect with the suffix *Instance*, such as ZoomInstance or FadeInstance.

When you use effects in your application, you are only concerned with the factory class; the instance class is an implementation detail. However, if you want to create custom effects classes, you must implement a factory class and an instance class. For more information, see Chapter 15, “Creating Effects,” in the *Creating and Extending Flex Components* book.

New Behaviors syntax

This section describes the syntax changes to behaviors.

The name property is now the id property

You now use the `id` property with effects, instead of the `name` property:

Flex 1.5:

```
<mx:Zoom name="small" duration="100"/>
```

Flex 2:

```
<mx:Zoom id="small" duration="100"/>
```

The <mx:Effect> tag is no longer necessary

The `<mx:Effect>` tag is no longer necessary in Flex:

Flex 1.5:

```
<mx:Effect>
   <mx:Zoom name="big" duration="100"/>
   <mx:Zoom name="small" duration="100"/>
</mx:Effect>
```

Flex 2:

```
<mx:Zoom id="big" duration="100"/>
<mx:Zoom id="small" duration="100"/>
```

Renamed the playEffect() and endEffect() methods

The `playEffect()` and `endEffect()` methods have been renamed. The new names are `play()` and `end()`.

Use binding in MXML to specify the effect

In Flex 1.5, you assigned the effect to an effect trigger property with no data binding:

```
<mx:Button id="myButton" creationCompleteEffect="myWL"/>
<mx:Button id="myOtherButton" creationCompleteEffect="myWL"/>
```

In Flex 2, you use data binding to assign an effect to a target:

```
<mx:Button id="myButton" creationCompleteEffect="{myWL}"/>
<mx:Button id="myOtherButton" creationCompleteEffect="{myWL}"/>
```

The show property is now the showTarget property

You now use the `MaskEffect.showTarget` property with effects, instead of the `MaskEffect.show` property:

Flex 1.5:

```
<mx:WipeUp id="wipeup" duration="1000" show="true"/>
```

Flex 2:

```
<mx:WipeUp id="wipeup" duration="1000" showTarget="true"/>
```

New properties added for the Zoom effect

The Zoom effect has new properties for Flex 2. The `zoomTo` property has been changed to `zoomHeightTo` and `zoomWidthTo`, and the `zoomFrom` property has been changed to `zoomHeightFrom` and `zoomWidthFrom`.

Also two more properties have been added to the Zoom effect:

originX, originY Specify the x-position and y-position of the origin, or *registration* point, of the zoom. The default value is the coordinates of the center of the effect target.

Change to the range of several effect properties

For the Zoom, Fade and Dissolve effects, the range of the `alpha`, `scaleX`, `scaleY`, `zoomHeightFrom`, `zoomWidthFrom`, `zoomHeightTo`, and `zoomWidthTo` properties have changed. You used to set these values as percentages using integer values, where 0 corresponded to 0%, and 100 to 100%. You now specify them as decimal values, where 0.0 corresponded to 0%, and to 1.0 corresponded to 100%.

Flex 1.5:

```
<mx:Effect>
   <mx:Zoom name="big" zoomTo="105" duration="100"/>
   <mx:Zoom name="small" zoomTo="100" duration="100"/>
</mx:Effect>
```

Flex 2:

```
<mx:Zoom id="big" zoomHeightTo="1.05" zoomWidthTo="1.05"duration="100"/>
<mx:Zoom id="small" zoomHeightTo="1.0" zoomWidthTo="1.0" duration="100"/>
```

Using the setStyle() method to set effects

The return value to the `getStyle()` method has changed when used with a behavior. Because trigger properties for behaviors are implemented as styles, you can use the `setStyle()` and `getStyle()` methods to manipulate triggers and their associated effects. The `setStyle()` method has the following signature:

```
setStyle("trigger_name", effect)
```

where:

trigger_name String indicating the name of the trigger property; for example, `mouseDownEffect` or `focusInEffect`.

effect The effect associated with the trigger. The data type of `effect` is a String containing the name of the effect, an Effect object, or an object of a subclass of the Effect class.

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

```
return_type getStyle("trigger_name")
```

where:

trigger_name String indicating the name of the trigger property.

return_type An Effect object, or an object of a subclass of the Effect class.

For detailed information, see Chapter 17, "Using Behaviors," in the *Flex 2 Developer's Guide*.

New events for effect classes

You can now associate event listeners with effects, rather than with effect targets. All effect classes now support the following event types:

`effectStart` Dispatched when the effect starts playing. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_START`.

`effectEnd` Dispatched after the effect stops playing, either when the effect finishes playing or when the effect has been interrupted by a call to the `endEffect()` method. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_END`.

Every effect class that is a subclass of the TweenEffect class, such as the Fade and Move effects, supports the following events:

`tweenStart` Dispatched when the tween effect starts. The type property of the event object for this event is set to TweenEvent.TWEEN_START.

`tweenEnd` Dispatched when the tween effect ends. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_END`.

`tweenUpdate` Dispatched every time a TweenEffect class calculates a new value. The `type` property of the event object for this event is set to `TweenEvent.TWEEN_UPDATE`.

Change to overriding the endEffect() method

You no longer have to call `listener.onEffectEnd()` in an override of the `endEffect()` method. Now, the `EffectInstance.endEffect()` method calls the instance class's `endEffect()` method, which calls the `EffectInstance.finishEffect()` method to dispatch the `EffectEvent.END_EFFECT` event and call `listener.onEffectEnd()`.

CHAPTER 9

Data Services

9

This topic describes how to migrate HTTPService, WebService, and RemoteObject components.

Contents

About Data Services

Configuration of data service components, now called Remote Procedure Call (RPC) components, in Flex has changed significantly in Adobe Flex 2.0. Adobe Flex Data Services now refers to the server-side feature set that includes RPC services, the Message Service, and the Data Management Service. Without Flex Data Services, you can use HTTPService and WebService tags, but you can access resources only on a server that is in the same domain as the Flex application or from a server that has a crossdomain.xml file installed on it. This file must allow access to the requesting application's domain. For more information about crossdomain.xml files, Chapter 4, "Applying Flex Security," in *Building and Deploying Flex Applications*.

For more information on using RPC components, see *Flex 2 Developer's Guide*.

Flex 2 separates the definitions of services into a new file, services-config.xml. This file contains definitions of the services and security constraints that were previously in the flex-config.xml file. Optionally, it can include other configuration files by reference. The services-config.xml file is located in the *flex_deploy_dir*/flex/WEB-INF/flex directory of a web application in which you are using Flex Data Services. If you use the command-line compiler, you must point to this file with the -services option.

The underlying architecture for communicating with the server for each type of service is based on a new messaging framework in Flex 2. As a result, you use message channels to communicate with the service. You configure channels in the services-config.xml file. The new new messaging framework uses channels to connect clients to endpoints; requests are made by sending messages over channels to endpoints of a message broker that directs the messages to the correct service.

For information about the client-side configuration of RPC components, see Chapter 45, "Using RPC Components," in the *Flex 2 Developer's Guide*. For information about the server-side configuration of RPC service destinations, see Chapter 46, "Configuring RPC Services" in the *Flex 2 Developer's Guide*.

Proxy use policy

In Flex 1.5, there was as setting that let you override the proxy. This was the `<proxy-use-policy>` setting in the flex-config.xml file. If you set it to `client`, Flex checked the value of the `useProxy` attribute on the service tag. The default was to use the proxy. If you set it to `always`, Flex used the proxy regardless of the value of the `useProxy` attribute. If you set it to `never`, Flex did not use the proxy regardless of the value of the `useProxy` attribute.

The `<proxy-use-policy>client</proxy-use-policy>` setting does not exist in Flex 2. Flex behaves as if the value is `client`, which means that Flex checks the value of the `useProxy` attribute on the service tag. The default value is `false`. If this value is not set, Flex does not use the proxy.

Channels

Flex now requires that you specify a channel to define the way data is transported for each RPC service destination. Service requests and responses are now messages. Messages are sent and received on a channel, which represents a logical connection to a destination. Channels define a protocol and a port.

Each channel corresponds to one network transfer protocol that Flash Player supports. For example, the AMF channel uses the AMF format over HTTP and the HTTP channel uses a text-based format over HTTP.

Channels are defined in the `channels` section of the services-config.xml file. There are several predefined channels that you can assign to your RPC service destinations. For more information about channels, see Chapter 43, "Configuring Data Services," in the *Flex 2 Developer's Guide*.

Logging

The `<web-service-proxy-debug>`, `<http-service-proxy-debug>`, and `<remote-objects-debug>` tags in flex-config.xml are no longer used in Flex 2. These were used for both client-side and server-side debugging. There is no complete replacement for these. There is a new client-side logging API; for more information, see Chapter 11, "Logging," in *Building and Deploying Flex 2 Applications*. There is also server-side logging that you can set in services-config.xml to log Remoting Service and Proxy Service traffic; for more information, see Chapter 43, "Configuring Data Services,"in the *Flex 2 Developer's Guide*.

Migrating RemoteObject components

The server-side configuration for RemoteObject components is now in the `<remoting-service>` section of the services-config.xml file. In Flex 1.5, you configured RemoteObjects in the `<remote-objects>` section of the flex-config.xml file.

To bind service results in Flex 2, you use the `lastResult` property of the service as the binding source, as the following example shows:

```
<mx:Text text="{tempService.getTemp.lastResult}"/>
```

In Flex 1.5, the `result` property was the binding source.

Unnamed RemoteObject

You can no longer use unnamed RemoteObjects. You must configure them in the `<remoting-service>` section in the services-config.xml configuration file or a file that it includes by reference. Adobe generally defines the Remoting Service in the remoting-config.xml file, which is included by reference in the services-config.xml file.

In Flex 1.5, you could specify `source="object_name"` and the object's statefulness on the RemoteObject tag. In Flex 2, you configure the object name and its statefulness in the configuration file.

The syntax for statefulness has changed. In Flex 1.5, you set the `type` to either `stateless-class` or `stateful-class`. In Flex 2, you set the `scope` attribute to `application`, `session`, or `request`. The default was `stateless-class`, and is now `request`, which is equivalent to stateless.

Flex 1.5 syntax

In Flex 1.5, you could specify the RemoteObject component's source and type in the MXML tag, while adding an entry in the `<remote-objects>` whitelist.

MXML tag:

```
<mx:RemoteObject id="MyService" source="credit.CreditCardAuth"
  type="stateless-class"/>
```

flex-config.xml file:

```
<remote-objects>
   <whitelist>
     <unnamed>
       <source>credit.*</source>
     </unnamed>
   </whitelist>
</remote-objects>
```

Flex 2 syntax

In Flex 2, every remote object must be configured as a Remoting Service destination in the services-config.xml file, or a file that it includes by reference, such as the remoting-config.xml file. You reference a destination in the `destination` property of the `<mx:RemoteObject>` tag.

MXML tag:

```
<mx:RemoteObject id="MyService" destination="SalaryEmployeeRO"/>
```

remoting-config.xml file:

```
<remoting-service>
  <destination id="SampleEmployeeRO" adapter="java-object">
    <properties>
      <source>samples.explorer.EmployeeManager</source>
      <scope>application</scope>
    </properties>
  </destination>
</remoting-service>
```

Named RemoteObject

This section describes how to migrate your named RemoteObject tags from Flex 1.5 to Flex 2.

Flex 1.5 syntax

In Flex 1.5, you used the `named` attribute to identify which named RemoteObject to use.

MXML tag:

```
<mx:RemoteObject id="employeeRO" named="SalaryRO">
   <mx:method name="getList"/>
</mx:RemoteObject>
```

flex-config.xml file:

```
<remote-objects>
   <whitelist>
      <named>
         <object name="SalaryRO">
            <source>samples.explorer.SalaryManager</source>
            <type>stateful-class</type>
         </object>
      </named>
   </whitelist>
</remote-objects>
```

Flex 2 syntax

In Flex 2, you use the `destination` attribute to identify which Remoting Service destination to use. In the configuration file, you define a destination.

MXML tag:

```
<mx:RemoteObject id="employeeRO" destination="SalaryEmployeeRO">
   <mx:method name="getList"/>
</mx:RemoteObject>
```

remoting-config.xml file:

```
<remoting-service>
  <destination id="SampleEmployeeRO">
    <properties>
      <source>samples.explorer.EmployeeManager</source>
      <scope>session</scope>
    </properties>
   </destination>
</remoting-service>
```

Migrating HTTPService components

This section describes how to migrate your HTTPService from Flex 1.5 to Flex 2.0 syntax.

For HTTPService tags, you specify the URL of the service in the `url` property of the tag.

The following examples shows and HTTPService tag that contacts a service directly:

```
...
<mx:HTTPService
  id="yahoo_web_search"
  url="http://api.search.yahoo.com/WebSearchService/V1/webSearch"
/>
...
```

You now configure HTTP services in the `<proxy-service>` section of the services-config.xml file or a file that it includes by reference. Adobe generally defines the Proxy Service in the proxy-config.xml file, which is included by reference in the services-config.xml file. Previously, it was configured in the `<http-service-proxy>` section of the flex-config.xml file.

To bind service results in Flex 2, you use the `lastResult` property of the service as the binding source, as the following example shows:

```
<mx:Text text="{yahoo_web_search.lastResult}"/>
```

In Flex 1.5, the `result` property was the binding source.

Unnamed HTTPService

This section describes how to migrate your unnamed HTTPService tags from Flex 1.5 to Flex 2.

For unnamed HTTPService tags that set `useProxy="false"`, the default value, you are not required to make any changes. These tags already ignore all server-side configuration.

Flex 1.5 syntax

In Flex 1.5, you added a URL pattern to the whitelist that matched the `url` of the HTTPService tag.

MXML tag:

```
<mx:HTTPService id="MyService" url="http://myServer.com/services/my.jsp"/>
```

flex-config.xml file:

```
<http-proxy>
   <whitelist>
      <unnamed>
         <url>http://myServer.com/services/*</url>
      </unnamed>
   </whitelist>
</http-proxy>
```

Flex 2 syntax

In Flex 2, the MXML tag syntax is almost the same, but you must also set the value of the `useProxy` property to `true`. Flex 2 does not support the `protocol` property that was available in Flex 1.5; the channel defines the protocol. Flex Data Services uses either the defaultHTTP or defaultHTTPS destination depending on whether the URL starts with HTTP or HTTPS, respectively.

In the services-config.xml file or a file that it includes by reference, such as the proxy-config.xml file, you add a `dynamic-url` to the `defaultHTTP` destination for an HTTPService. The URL pattern must match the URL used in the MXML tag.

MXML tag:

```
<mx:HTTPService id="MyService" url="http://myServer.com/services/my.jsp"
  useProxy="true"/>
```

proxy-config.xml file:

```
<destination id="defaultHTTP">
   <properties>
      <dynamic-url>http://myServer.com/services/*</dynamic-url>
         ...
   </properties>
</destination>
```

Named HTTPService

This section describes how to migrate named HTTPService tags from Flex 1.5 to Flex 2 syntax.

Flex 1.5 syntax

In Flex 1.5, a named HTTPService was defined as a named whitelist entry in the `<http-service-proxy>` section. You used the `serviceName` attribute of the HTTPService tag to identify it.

MXML tag:

```
<mx:HTTPService id="MyService" serviceName="Salary" protocol="https"/>
```

flex-config.xml file:

```
<http-service-proxy>
   <whitelist>
      <named>
         <service name="Salary">
            <url>https://www.myServer.com/services/salary.jsp</url>
         </service>
      </named>
   </whitelist>
</http-service-proxy>
```

Flex 2 syntax

In Flex 2, you use the `destination` attribute of the HTTPService tag to identify the named service in your MXML files. You also must set the value of the `useProxy` property to `true`. Flex 2 does not support the `protocol` property; the channel defines the protocol.

MXML tag:

```
<mx:HTTPService id="employeeHTTP" destination="Salary" useProxy="true"/>
```

proxy-config.xml file:

```
<proxy-service>
   <destination id="Salary">
      <properties>
         <url>https://www.myServer.com/services/salary.jsp</url>
      </properties>
   </destination>
</proxy-service>
```

Migrating WebService components

The default value of the `useProxy` property is `false`. The WebService tags are now configured in the `<proxy-service>` section of the services-config.xml file or a file that it includes by reference. They were previously described in the `<web-service-proxy>` section of the flex-config.xml file.

To bind service results in Flex 2, you use the `lastResult` property of the service as the binding source, as the following example shows:

```
<mx:Text text="{tempService.getTemp.lastResult}"/>
```

In Flex 1.5, the `result` property was the binding source.

Unnamed WebService

This section describes how to migrate unnamed WebService tags from Flex 1.5 to Flex 2.

For unnamed WebServices tags that set `useProxy="false"`, you are not required to make any changes. These tags already ignore all server side configuration.

Flex 1.5 syntax

In Flex 1.5, you added an unnamed whitelist entry to the `<web-service-proxy>` section of the configuration file.

MXML tag:

```
<mx:WebService id="MyService" wsdl="http://myServer.com/services/my.wsdl"/>
```

flex-config.xml file:

```
<web-service-proxy>
  <whitelist>
     <unnamed>
       <url>http://myServer.com/services/*</url>
     </unnamed>
  </whitelist>
</web-service-proxy>
```

Flex 2 syntax

In Flex 2, the MXML tag syntax for an unnamed WebService is almost the same, but you must also set the value of the `useProxy` property to `true`. Flex 2 does not support the `protocol` property that was available in Flex 1.5; the channel defines the protocol. In the configuration file, you must specify a channel and an adapter that matches the value of the `wsdl` attribute.

MXML tag:

```
<mx:WebService id="MyService" wsdl="http://myServer.com/services/my.wsdl"
  useProxy="true"/>
```

services-config.xml file:

```
<destination id="defaultHTTP">
  <properties>
    <wsdl>{context.root}/services/ContactManagerWS?wsdl</wsdl>
    <soap>{context.root}/services/ContactManagerWS</soap>
  </properties>
  <adapter ref="soap-proxy"/>
</destination>
```

Named WebService

This section describes how to migration your named WebService tags from Flex 1.5 to Flex 2.

Flex 1.5 syntax

In Flex 1.5, you referred to a named WebService with the `serviceName` attribute of the `<mx:WebService>` tag. In the configuration file, you defined a service's WSDL and endpoint as entries in the whitelist.

MXML tag:

```
<mx:WebService id="employeeWS" serviceName="SalaryWS">
   <mx:operation name="getList"/>
</mx:WebService>
```

flex-config.xml file:

```
<web-service-proxy>
   <whitelist>
      <named>
      <service name="SalaryWS">
         <wsdl>{context.root}/services/SalaryWS.wsdl</wsdl>
         <endpoints>
            <endpoint>{context.root}/services/SalaryWS</endpoint>
         </endpoints>
      </service>
   </whitelist>
</web-service-proxy>
```

Flex 2 syntax

In Flex 2, you refer to named WebServices with the `destination` attribute of the `<mx:WebService>` tag. In the configuration file, you define the location of the WSDL file and the endpoint as part of the service destination.

MXML tag:

```
<mx:WebService id="employeeWS" destination="SalaryWS">
   <mx:operation name="getList"/>
</mx:WebService>
```

services-config.xml file:

```
<proxy-service>
  <destination id="SalaryWS">
    <properties>
      <wsdl>{context.root}/services/SalaryWS?wsdl</wsdl>
      <soap>{context.root}/services/SalaryWS</soap>
    </properties>
    <adapter ref="soap-proxy"/>
  </destination>
</proxy-service>
```

Migrating secure data services

This section describes the changes you must make to your destination definitions in order to use secured data services in your Flex applications.

Migrating services that use run-as

Third-party service endpoints may require authentication information. In Flex 1.5, you used the `run-as` element to pass credentials to remote endpoints. In Flex 2, you use `remote-username` and `remote-password` elements to specify credentials that a remote endpoint requires.

The changes in this section apply to the HTTPService, WebService, and RemoteObject services.

Flex 1.5 syntax

In Flex 1.5, you specified run-as credentials as `user` and `password` attributes to pass through credentials to a service.

flex-config.xml file:

```
<web-service-proxy>
   <whitelist>
     <named>
       <service name="MyService">
         <wsdl>http://somewhere.com/webservice.wsdl</wsdl>
         <endpoint>http://somewhere.com/myservice</endpoint>
         <run-as user="user1" password="opensaysme"/>
       </service>
     </named>
   </whitelist>
</web-service-proxy>
```

Flex 2 syntax

In Flex 2, you set the `remote-username` and remote-password elements in the destination definition.

services-config.xml file:

```
<destination id="samplesProxy">
  <properties>
    <url>
      http://someserver/SecureService.jsp
    </url>
    <remote-username>johndoe</remote-username>
    <remote-password>opensaysme</remote-password>
  </properties>
</destination>
```

Alternatively, you can pass remote credentials from an RPC component in the component's `setRemoteCredentials(`*`remoteUsername, remotePassword`*`)` method from the client at run time.

Migrating services that use Basic authentication

For Basic authentication, you must change the value of the security constraint's `url-pattern` in the web.xml file. This section assumes that you have already migrated the service's destination, as described in previous sections.

Flex 1.5 syntax

In Flex 1.5, for WebService and HTTPService, you specified the /flashproxy/*service_name* as the `url-pattern`, as the following example shows:

```
<web-app>
   ...
   <security-constraint>
     <web-resource-collection>
       <web-resource-name>Protected Page</web-resource-name>
       <url-pattern>/flashproxy/MyService</url-pattern>
       <http-method>GET</http-method>
       <http-method>POST</http-method>
     </web-resource-collection>
     <auth-constraint>
       <role-name>manager</role-name>
     </auth-constraint>
   </security-constraint>

   <security-role>
     <role-name>manager</role-name>
   </security-role>
   <login-config>
     <auth-method>BASIC</auth-method>
   </login-config>
</web-app>
```

For RemoteObject, you specified /amfgateway/*service_name* as the `url-pattern`.

Flex 2 syntax

In Flex 2, you specify the URI of the channel endpoint for which you want to require authentication as the `url-pattern`. You use the value of the channel definition's endpoint URI in the services-config.xml file. The boldface text in the following example shows a URI that requires authentication:

```
<web-app>
   ...
   <security-constraint>
     <web-resource-collection>
       <web-resource-name>Protected Channel</web-resource-name>
       <url-pattern>/messagebroker/amf</url-pattern>
       <http-method>GET</http-method>
       <http-method>POST</http-method>
     </web-resource-collection>
     <auth-constraint>
       <role-name>manager</role-name>
     </auth-constraint>
   </security-constraint>
   <security-role>
     <role-name>manager</role-name>
   </security-role>
   <login-config>
     <auth-method>BASIC</auth-method>
   </login-config>
</web-app>
```

Migrating services that use custom authentication

This section describes how to migrate a RemoteObject tag that used custom authentication from Flex 1.5 to Flex 2. This section assumes that you have already migrated the named RemoteObject as described in "Named RemoteObject" on page 139.

You could previously only use custom authentication with named RemoteObjects. Now you can use custom authentication with all service types. You do not modify web.xml to lock down a URL in this sort of authentication.

Flex 1.5 syntax

In Flex 1.5, you specified the type of authentication and the role in the named service definition in the flex-config.xml file:

```
<named>
   <object name="myobj">
      <use-custom-authentication>true</use-custom-authentication>
      <roles>
         <role>sampleusers</role>
      </roles>
   </object>
</named>
```

Flex 2 syntax

In Flex 2, you define the security constraint in the services-config.xml file:

```
<service-config>
   <services>
      ... // Destinations are defined here.
   </services>
   <security>
      <security-constraint id="sample-users">
         <auth-method>Custom</auth-method>
         <roles>
            <role>sampleusers</role>
         </roles>
      </security-constraint>
   </security>
</service-config>
```

You then refer to that security constraint in your destination definition, which is also in the services-config.xml file or a file that it includes by reference:

```
<destination>
   ...
   <security>
      <security-constraint ref="sample-users"/>
   </security>
</destination>
```

Login commands

For custom authentication, Flex uses a custom login adapter, known as a login command, to check a principal's credentials and let that principal log into the application server. A login command must implement the flex.messaging.security.LoginCommand API.

Flex 1.5 and Flex 2 include default login command implementations for Adobe JRun, BEA WebLogic, IBM WebSphere, Apache Tomcat, and Oracle. In Flex 1.5, these were stored in the gateway-config.xml file. You could add your own custom login commands to that file.

In Flex 2, the gateway-config.xml file no longer exists. The default login commands are now in the `<security>` block of the services-config.xml file. To migrate custom login commands, you must move them to this location. Use the TomcatLoginCommand class for either Tomcat or JBoss.

The following example shows the `<security>` section of the services-config.xml file. This is where you move custom login commands. You should enable only one login command at a time; comment out all others.

```
<security>
   <login-command class="flex.messaging.security.JRunLoginCommand"
     server="JRun"/>
   <!--
   <login-command class="flex.messaging.security.TomcatLoginCommand"
     server="Tomcat"/>
   <login-command class="flex.messaging.security.WeblogicLoginCommand"
     server="Weblogic"/>
   <login-command class="flex.messaging.security.WebSphereLoginCommand"
     server="WebSphere"/>
-->
</security><>
```

Mapping Java types for RemoteObject

When an ActionScript type is not handled implicitly, you can map it to a typed Java class of the same name on the server.

In Flex 1.5, you create a static variable in the ActionScript class that uses the `Object.registerClass()` method to specify the fully qualified name of the corresponding Java class on the server. The first parameter of the `registerClass()` method is the fully qualified name of the Java class; the second parameter is the fully qualified name of the ActionScript class.

`Object.registerClass()` is not available in Flex 2. ActionScript 3 provides the flash.net.registerClassAlias. To simplify using this class Flex 2, you can specify a remote class in the `[RemoteClass(alias="`*`remoteclassname`*`")]` metadata tag above the class definition in your ActionScript class.

Flex 1.5 ActionScript class example:

```
class com.Product {
   public var id:Number;
   public var name:String;
   public var price:Number;
   public var description:String;
   public static var regClass = Object.registerClass("com.Product",
     com.Product);

   public function Product() {
   }

   public function toString():String {
     return "id = " + id + " name = " + name + " price = $" + price;
   }
}
```

In Flex 2, your code should look like this:

```
package samples.customer
{
  [RemoteClass(alias="samples.customer.Customer")]
  public class Customer {
    public var custId:int;
    public var firstName:String = "";
    public var lastName:String = "";
    public var cellPhone:String = "";
    public var email:String = "";
    public var partySize:int;
    public var tableReady:Boolean = false;
  }
}
```

Accessing request/response data with RemoteObject

A Java object that you call using the `<mx:RemoteObject>` tag has access to request, response, and servlet data. For Flex 1.5, from within a Java object, you can call the following methods:

Method	Description
`flashgateway.Gateway.getHttpRequest()`	Returns the HttpServletRequest object for the current request. Adobe recommends that you access session data and other request data through the `getHttpRequest()` method.
`flashgateway.Gateway.getHttpResponse()`	Returns the HttpServletResponse object for the current request.
`flashgateway.Gateway.getServletConfig()`	Returns the ServletConfig object for the calling servlet.

In Flex 2, this API no longer exists. The flex.messaging.FlexContext class provides equivalent methods. Flex Data Services provides enhancements for working with session data in the following classes:

- flex.messaging.FlexContext
- flex.messaging.FlexSession
- flex.messaging.FlexSessionListener
- flex.messaging.FlexSessionAttributeListener
- flex.messaging.FlexSessionBindingEvent
- flex.messaging.FlexSessionBindingListener

These classes are included in the the public Flex Data Services Javadoc documentation. For information, see Chapter 43, “Configuring Data Services,” in the *Flex 2 Developer’s Guide*.

CHAPTER 10

Configuration and Command Line Tools

10

The flex-config.xml file has undergone significant changes for Adobe Flex 2. In addition, the mxmlc, compc, and fdb utilities are also changed. This topic describes these changes.

Contents

Configuration files

The Flex server relied on configuration files in the WEB-INF/flex directory. The following table describes changes to those configuration files:

Configuration file	Flex 1.5	Flex 2
flash-unicode-table.xml	Lists convenient mappings of the Flash MX 2004 UnicodeTable.xml character ranges for use in the Flex configuration file.	No changes.
flex-config.xml	Configures Flex. You use this file to define debugging, compiler, cache, proxy, logging, font and other settings for Flex.	This file now contains only compiler settings.
gateway-config.xml	Configures the Adobe Flash Remoting gateway. You can configure service adapters, security, logging, and other settings for Flash Remoting using this file.	Not in Flex 2. The settings have been move to the services-config.xml file. For more information, see Chapter 9, "Data Services," on page 135.
global.css	Defines default styles used across all Flex applications.	This file has been replaced with the defaults.css file inside the frameworks.swc file. It is not intended for developers to edit, but can be used as a template from which to define custom themes.
license.properties	Stores license key.	Not in Flex 2.
mxml-manifest.xml	Map components to namespaces.	Not in Flex 2.

The following configuration files were renamed since Flex 2 Beta 2:

Flex 1.x Name	Flex 2 Name
flex-enterprise-services.xml, became fds-config.xml	services-config.xml
flex-data-service.xml, became fds-data-management.xml,	data-management-config.xm
flex-message-service.xml, became fds-messaging.xml,	messaging-config.xml
flex-remoting-service.xml, became fds-remoting.xml,	remoting-config.xml
flex-proxy-service.xml, became fds-proxy.xml,	proxy-config.xml

Security

This section describes changes to Flex security.

Flex changes

You do not define the security of web services and other data services in the flex-config.xml file. You now define them in the services-config.xml and related files.

ActionScript changes

The following general changes were made to ActionScript:

- The System.security.* package is now named Security.*.
- The `Security.allowDomain()` and `Security.allowInsecureDomain()` methods no longer open up all SWF files in the caller's domain; instead they now affect only the calling SWF file itself.
- Setting Security.exactSettings no longer affects all SWF files in the caller's domain; instead it now affects only the calling SWF file itself.
- The `LocalConnection allowDomain()` and `allowInsecureDomain()` methods are no longer callback methods for authors to define; instead they are now built-in methods for authors to call, and follow the same semantics as the `Security.allowDomain()` and `Security.allowInsecureDomain()` methods.
- When you use XMLSocket.connect to contact a server outside a SWF file's own domain, the default policy file location is no longer on an HTTP server in the same domain; instead it is now an XMLSocket policy file obtained from the same port as the main connection attempt. You can use the `Security.loadPolicyFile()` method to override this default location in the same way you used it in Flex 1.5.

Command-line compilers

The mxmlc and compc compilers are changed for Flex 2. This section describes these changes.

mxmlc

The mxmlc utility compiles SWF files from your MXML and ActionScript files. The mxmlc options that are no longer available include the following:

- `batch`
- `contextroot`
- `encoding`
- `genlibdir`
- `headless`
- `loglevel`
- `profile`
- `file-specs`
- `systemclasses`
- `version`
- `webroot`

The names of many mxmlc command-line compiler options were changed to be more consistent or descriptive. The changes include the following:

Flex 1.x Name	Flex 2 Name	Comment
`aspath` became `actionscript-classpath`	`source-path`	
`configuration`	`config`	
`debugpassword`	`debug-password`	
`global-css-url`	`defaults-css-url`	
`g`	`debug`	No longer generates a SWD file. The resulting SWF file contains the debug code. Also, there is no longer a framework_debug.swc file. All the debug logic is built into the framework.swc file. As a result, you no longer need to specify a value for the `debug-library-path` option. This option was removed.
`libpath`	`library-path`	

Flex 1.x Name	Flex 2 Name	Comment
`O (optimize)`	`optimize`	The `optimize` option no longer suppresses `trace()` method output. You must manually remove the output.
`report`	`link-report`	
`usenetwork`	`use-network`	

In addition, because the data services subsystem changed, mxmlc no longer takes the following options:

- `gatewayurl`
- `gatewayhttpsurl`
- `proxyurl`
- `proxyhttpsurl`
- `proxyallowurloverride`
- `remoteallowurloverride`
- `webserviceproxydebug`

The maximum value for the `default-script-limits` option is now 60 seconds. This option did not impose a maximum value in Flex 1.x.

The `file-specs` option has been removed. You do not need to specify `-file-specs=filename.mxml` because it is the default option. If the last option uses a space-separated list, you can terminate the list with `--` before adding the MXML file name; for example:

```
mxmlc -option arg1 arg2 arg3 -- MyApp.mxml
```

For a complete list of the mxmlc options, see *Building and Deploying Flex 2 Applications*.

compc

The compc utility compiles SWC files. The compc parameters have changed significantly. For usage information, see the command-line help or the *Building and Deploying Flex 2 Applications*.

SWC files created by the Flex 1.x compiler or by previous versions of Flash do not work in Flex 2. If you try to use a SWC file that was generated by an earlier version of compc or output from the Flash IDE, Flex displays a compiler error similar to the following:

```
Unable to parse SWC catalog for C:\JRun4\servers\flex2\flex\WEB-
  INF\flex\user_classes\ModalText.swc: Unknown element in swc section in
  catalog.xml: componentPackage
```

fdb debugger

This section describes changes to the fdb debugger. For more information on using fds, see Chapter 12, "Using the Command-Line Debugger," in *Building and Deploying Flex 2 Applications*.

SWD files

The fdb debugger no longer uses SWD files. Instead, Flex generates debuggable SWF files by using the `debug` option with the mxmlc and compc command line compilers. There is no longer a framework_debug.swc file. All the debug logic is built into the framework.swc.

Breakpoints

This section describes changes to breakpoints for Flex 2.

Player lock out

When you encounter a breakpoint with the fdb debugger, Flash Player locks out user interaction. You cannot click on anything playing or access its menu options; you also cannot close Flash Player and it does not redraw its display.

The reason for this behavior is to make breakpoints more predictable. For example, when you are locked out of Flash Player, no other messages such as focus-change messages or user interactions can change the results of the debugging session.

Deferred breakpoints

The fdb utility now supports deferred breakpoints. What this means is that if you try to set a breakpoint, and the breakpoint location that you specified seems to indicate a filename or function name that is not yet loaded into the Flash player, the fdb no longer reports an error; instead, it remembers the breakpoint. If Flash Player eventually loads a movie that has code from the source file that was specified in the earlier break command, fdb adds the breakpoint.

run command

On Windows, you can enter either `run foo.swf`, in which case fdb launches Flash Player, or `run`, in which case fdb displays the "Waiting for Player to connect" message; at that point, you must manually launch Flash Player.

On the Macintosh, the `run foo.swf` command is no longer supported; fdb cannot launch Flash Player. The only supported form of the command on the Macintosh is `run`, after which you must manually launch Flash Player.

print command

Flex 2 includes the following changes to the output of the print command:

- If a member of an object is an integer, its value is printed in both decimal and hexadecimal.
- If `set $displayattributes` was specified before the `print` command is executed, the list of attributes that are displayed is different. The new list of attributes is:
  - `dont_enumerate`
  - `read_only`
  - `local`
  - `argument`
  - `getter`
  - `setter`
  - `dynamic`
  - `static`
  - `private`
  - `public`
  - `internal`
  - `has_namespace`

Commands no longer supported

The following features of the fdb debugger in Flex 1.5 are no longer supported in Flex 2:

- Watchpoints: the `watch`, `awatch`, and `rwatch` commands.
- Disassembly: the `disassemble` command.
- On the Macintosh only, the `run` command.

CHAPTER 11

Customizing Components

11

This topic describes modifications to the process of creating components in ActionScript in Adobe Flex.

This topic only contains an overview of the major changes to the process from Flex 1.5 to Flex 2.0. For detailed information on creating custom components, see *Creating and Extending Flex 2 Components*.

Contents

UIObject class removed

The UIObject class has been removed for Flex 2.

Class variables changed

Remove the `symbolName` and `symbolOwner` class variables. Symbols are no longer important because display objects can be instantiated using the new operator.

Specifying the package

Define your custom components within an ActionScript package. The package reflects the directory location of your component within the directory structure of your application.

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

Defining the class

The class definition must be prefixed by the `public` keyword, as the following example shows:

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

  // Define properties, constructor, and methods.

}
```

Defining the constructor

If the class is missing a constructor, add it. A constructor for a child class of UIComponent must have no required arguments; it can only have optional ones.

Here is a typical constructor:

```
public function Button() {
   super();
   className = "Button";
   btnOffset = 0;
}
```

Creating bindable properties

In Flex 1.5, you use the `[ChangeEvent]` metadata tag to define a property as bindable. In Flex 2.0, you use the `[Bindable]` metadata tag. For more information, see *Creating and Extending Flex 2 Components*.

Overriding a method

If the method is overriding a method in a superclass, add the `override` keyword as the first attribute:

```
override public function createChildren():void
```

If the getter/setter is overriding a getter/setter in a superclass, add the `override` keyword as the first attribute:

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

By convention, setters should use the identifier `value` for their argument.

Clip parameters removed

Remove anything related to the `clipParameters` variable.

Initialization sequence changed

In Flex 1.5, the component initialization sequence was as follows:

1. Constructor
2. `init()`
3. `createChildren()`
4. `commitProperties()`
5. `measure()`
6. `layoutChildren()`
7. `draw()`

In Flex 2.0, the `init()` method has been removed, and the new `updateDisplayList()` method replaces the `layoutChildren()` and `draw()` methods. You can move logic that was formerly in the `init()` method to the constructor. For a complete description of the initialization sequence for Flex 2, see Chapter 10, "Creating Advanced Visual Components in ActionScript," in *Creating and Extending Flex 2 Components*.

Renamed invalidateStyle()

The `invalidateStyle()` method has been renamed to `styleChanged()`.

CHAPTER 12

Additional Migration Issues

12

This topic describes miscellaneous migration issues, including charting and Runtime Shared Libraries (RSLs).

Contents

HistoryManager

To migrate applications from Flex 1.5 to Flex 2, your application must implement the mx.core.IHistoryState interface; for example:

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

Also, you must declare the `saveState()` and `loadState()` methods as `public`. Otherwise, they are not visible to the HistoryManager.

Charting

Flex Charting have undergone major changes for Flex 2. This section describes some migration issues for charts for Flex 2.

Charting classes rely heavily on DataProviders and Arrays. Changes to these classes are described throughout this document.

The `name` property on chart series and chart axes is now `displayName`.

Skins

Skins are now called renderers. For example, the CandlestickSkin class is now CandlestickRenderer.

Renderers

Some functionality for the AxisRenderer was moved to other axis objects.

The `labelFunction` property, which was a property of AxisRenderer, is now a property of the axis type (such as CategoryAxis). The signature for the function is changed as well. Instead of a single parameter, `the labelFunction()` function now takes up to four. The new signature is as follows:

```
labelFunction(categoryValue:Object, previousCategoryValue:Object,
  axis:axis_type, categoryItem:Object);
```

For more information on using the `labelFunction` property, see the *Flex 2 Developer's Guide*.

The `title` property was also moved to the individual axis rather than the axis renderer. For example, in Flex 1.x you defined both the horizontalAxis and the horizontalAxisRenderer, as the following example shows:

```
<mx:horizontalAxis>
  <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month"/>
</mx:horizontalAxis>
<mx:horizontalAxisRenderer>
  <mx:AxisRenderer title="Expenses" labelFunction="defineLabel"/>
</mx:horizontalAxisRenderer>
```

In Flex 2, you set the `title` and `labelFunction` properties on the horizontalAxis:

```
<mx:horizontalAxis>
  <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month"
    title="Expenses" labelFunction="defineLabel"/>
</mx:horizontalAxis>
```

You no longer use renderers to change the appearance of ChartItems. In Flex 1.x, for example, you could specify a CrossRenderer or TriangleRenderer to draw a ChartItem as a cross or a triangle:

```
<mx:PlotSeries>
  <mx:renderer>
    <mx:CrossRenderer/>
  </mx:renderer>
</mx:PlotSeries>
```

In Flex 2, you set the value of the series' `itemRenderer` property to a skin class that draws the ChartItem's icon:

```
<mx:PlotSeries itemRenderer="mx.charts.skins.halo.CrossSkin"/>
```

In addition, you can no longer use the AssetRenderer to use graphics in your charts. Instead, you must use a class that implements the IDataObject interface.

Binding

In many cases, you declare a data provider object and bind the chart to that data provider. In Flex 2, you must add the `[Bindable]` metadata tag to the variable declaration, otherwise Flex does not bind to the data. For example:

```
<mx:Script>
  [Bindable]   // Add this in Flex 2
  public var expenses:Object = [{ ... }, { ... }, { ... }];
</mx:Script>
<mx:BubbleChart maxRadius="50" dataProvider="{expenses}"
```

mouseDown events

In Flex 1.5, `mouseDown` events included `hitData` structures, even if no data was under the mouse's pointer. In this case, the `hitData` property existed, but it was `null`. This behavior was a way to check for the existence of a click on a chart control.

In Flex 2, `mouseDown` events do not include the `hitData` structure unless the mouse is positioned over a data point. Instead, you must use the `mouseDownData` event.

alpha

All aspects of charting that used an alpha property to represent transparency, such as Strokes and Fills, now use 0 to 1 for a range of values rather than 1 to 100. For example:

Flex 1.x:

```
<mx:SolidColor color="0x7EAEFF" alpha="30"/>
```

Flex 2:

```
<mx:SolidColor color="0x7EAEFF" alpha=".3"/>
```

Legends

You now enclose the data provider for Legend controls in curley braces; for example:

Flex 1.x:

```
<mx:LineChart id="linechart">
   ...
</mx:LineChart>
<mx:Legend dataProvider="linechart"/>
```

Flex 2:

```
<mx:LineChart id="linechart">
   ...
</mx:LineChart>
<mx:Legend dataProvider="{linechart}"/>
```

Interfaces

All chart interfaces now follow the I* naming scheme, as the following table shows:

Flex 1.x Name	Flex 2 Name
BoxRenderer	IBoxRenderer
AreaRenderer	IAreaRenderer
WedgeRenderer	IWedgeRenderer
LineRenderer	ILineRenderer
Fill	IFill
Axis	IAxis
AxisRenderer	IAxisRenderer (the interface, not the class)

Cell renderers

In Flex 1.5, a cell renderer had to implement the `setValue()` method to access the data passed to the cell renderer:

```
<mx:VBox xmlns:mx="http://www.adobe.com/2003/mxml" width="198"
  backgroundAlpha="0">
<mx:Script>
  function setValue(str:String, item:Object) {
    // Set values of the cell renderer controls.
  }
</mx:Script>
  <mx:HBox height="100%">
    <mx:Image id="myImage" width="30"/>
    <mx:Text text="{labelData}" width="150" height="100%"/>
  </mx:HBox>
  <mx:CheckBox label="Check"/>
  <mx:TextInput/>
</mx:VBox>
```

In Flex 2, cell renderers and cell editors were renamed to item renderers and item editors. In addition, the architecture was completely redesigned for Flex 2. For more information, see Chapter 21, "Using Item Renderers and Item Editors," in the *Flex 2 Developer's Guide*.

In Flex 2, cell renderers receive a `data` property that contains the data for the item to render. For example, for a cell of a DataGrid control, the `data` property contains a copy of the data provider element for the entire row of the grid. You access the `data` property in your cell renderer to initialize it, as the following example shows:

```
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml" width="198"
  backgroundAlpha="0">
  <mx:HBox height="100%">
    <mx:Image id="myImage" source="{data.imageLocal}" width="30"/>
    <mx:Text text="{data.displayText}" width="150" height="100%"/>
  </mx:HBox>
  <mx:CheckBox label="Check"  selected="{data.status}"/>
  <mx:TextInput selected="{data.moreText}"/>
</mx:VBox>
```

In this example, you use data binding to set the values of the controls in the cell renderer, where the fields of the `data` property correspond to fields of the data provider that populates the DataGrid control.

Validators

In Flex 1.5, validators were usually triggered in response to an update of the destination of a data binding expression. You typically assigned the validator to the destination of a data binding expression, and triggered the validation when the destination of the data binding expression was updated, as the following example shows:

```
<!-- Define a data model for storing the phone number. -->
<mx:Model id="userInfo">
  <phoneNum>{phoneInput.text}</phoneNum>
</mx:Model>

<!-- Define the PhoneNumberValidator. -->
<mx:PhoneNumberValidator field="userInfo.phoneNum"/>

<!-- Define the TextInput control for entering the phone number. -->
<mx:TextInput id="phoneInput"/>
```

In Flex 2, validators are triggered by default by the `valueCommit` event, and you use data binding to assign it to the interface control rather than to a model. The source property specifies the name of the control, and the `property` property specifies the field of the control to validate, as the following example shows:

```
<!-- Define the PhoneNumberValidator. -->
<mx:PhoneNumberValidator id="pnV" source="{phoneInput}" property="text" />

<!-- Define the TextInput control for entering the phone number. -->
<mx:TextInput id="phoneInput"/>
```

You can also use the `validate()` method of the validator to trigger a validator programmatically. All validator classes now include a `validate()` method.

CreaditCardValidator constants moved to a new class

The constants that define the type of credit card to validate were moved from the CreditCardValidator class to a new class named CreditCardValidatorCardType.

The constants were also renamed. In MXML, valid constants values are:

- `"American Express"`
- `"Diners Club"`
- `"Discover"`
- `"MasterCard"`
- `"Visa"`

In ActionScript, you can use the following constants:

- `CreditCardValidatorCardType.AMERICAN_EXPRESS`
- `CreditCardValidatorCardType.DINERS_CLUB`
- `CreditCardValidatorCardType.DISCOVER`
- `CreditCardValidatorCardType.MASTERCARD`
- `CreditCardValidatorCardType.VISA`

Deprecated methods, properties, and events

The following validator methods, properties, and events were deprecated:

Deprecated item	New item
`Validator.enable()` and `Validator.disable()` methods	`Validator.enabled` property
`Validator.isValid()` and `Validator.isStructureValid()`	`Validator.validate()` method
`Validator.hasErrors()`	You now examine the Event object from the validation to determine if any errors occurred.
`Validator.field`	You now use `Validator.source` and `Validator.property` to specify the item to validate.
`UIComponent.validationFailed` and `UIComponent.validationSucceeded` events	`UIComponent.valid` and `UIComponent.invalid` events, or `Validator.valid` and `Validator.invalid` events

Styles

When you change a validator's error message style, you must now use a class selector rather than a type selector. You do this by prepending a period to the `errorTip` style as the following example shows:

```
<mx:Style>
   // ErrorTip { borderColor: #00FFFF } // Flex 1.5
   .errorTip { borderColor: #00FFFF } // Flex 2
</mx:Style>
```

Embedding resources

In Flex 1.5, embedded resources were bound to Strings that were used to reference the individual images by name. Although the preferred method of embedding resources in Flex 2 uses Class variables, you can still use String variables for some level of backward compatibility. However, the various objects and tags that use your embedded assets expect them to be tied to Class variables, so you need to use the `getDefinitionByName()` method to cast your string variables. You also still need to use the `[Bindable]` metadata tag to declare the string variable bindable.

For instance, the following application uses string variables and casting to display an embedded image:

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

  <mx:Script>
    <![CDATA[
      [Embed(source="logo.gif")]
      [Bindable]
      public var imgStr:String;
    ]]>
  </mx:Script>

  <mx:Image source="{getDefinitionByName(imgStr)}"/>
</mx:Application>
```

Although this method works, Adobe recommends that you use Class variables instead. The equivalent application, using a Class variable, is simpler:

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

  <mx:Script>
    <![CDATA[

      [Embed(source="logo.gif")]
      [Bindable]
      public var imgCls:Class;
    ]]>
  </mx:Script>

  <mx:Image source="{imgCls}"/>
</mx:Application>
```

Alternatively, when you use MXML, this becomes a one line application:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Image source="@Embed(source='logo.gif')" />
</mx:Application>
```

For more information, see Chapter 30, “Embedding Assets,” in the *Flex 2 Developer’s Guide*.

CHAPTER 13

Migration Patterns

13

This topic describes some common patterns in migrating ActionScript in Adobe Flex applications.

Contents

Instantiating Flex controls

In Flex 1.x, you created a Flex control in ActionScript by first including a reference to that control, and then using the `createEmptyObject()`, `createChild()`, `createChildren()`, `createChildAtDepth()`, or `createClassChildAtDepth()` method.

These methods were removed. In Flex 2, you use the `new` operator to create child controls and attach the control to a container with the `addChild()` or `addChildAt()` method. For example:

Flex 1.x:

```
var b:Button;
b = Button(createChild(Button, undefined, { label: "OK" }));
```

Flex 2:

```
var b:Button = new Button();
b.label = "OK";
addChild(b);
```

Similarly, in Flex 2 you would destroy the object with the `destroyObject()`, `destroyChild()`, `destroyChildAt()`, or `destroyAllChildren()` method. These methods are also deprecated. Instead, you use the `removeChild()` or `removeChildAt()` method.

Flex 1.x	**Flex 2**
`createComponent()` method	`createComponentFromDescriptor()`
`createComponents()` method	`createComponentsFromDescriptors()`

The `createComponent()` method now takes only a descriptor as its first argument, rather than either a descriptor or a descriptor index. If you know the index, use `childDescriptors[i]` to get the descriptor itself.

For more information on creating and destroying Flex controls in ActionScript, see the *Flex 2 Developer's Guide*.

Using mixins

You can no longer attach a function to a class, unless that class has prior knowledge of that function. For example, you can no longer do this:

```
UIComponent.prototype.doSomething = myFunction
```

or this:

```
dataGridInstance.doSomething = myFunction
```

You can still declare a Function type property on an Object and then supply an implementation of that function later. For example:

```
class MyButton extends Button {
  var doSomething:Function;
  public function processInput(condition:Boolean):void {
  if (condition)
    doSomething();
  }
}
```

and then:

```
var b:MyButton = new MyButton();
b.doSomething = function () { ... };
b.processInput(true);
```

You can also apply mixins to dynamic classes without their prior knowledge, as the following example shows:

```
dynamic class MyButton extends Button {
  ...
}
```

and then:

```
// You can mix in any function onto an instance of a dynamic class:
var b:MyButton = new MyButton();
b.anyFunctionNameYouCanImagine = function () { ... };

// After it's added, you can call the function as follows:
b.anyFunctionNameYouCanImagine();
```

The only class in the Flex class library that is dynamic is the Object class. In most cases, you must create your own class.

Variable enumeration and object introspection

In Flex 1.x (ActionScript 2.0), using a `for-in` loop on an object let you enumerate over all properties on an object. In Flex 2 (which uses ActionScript 3.0), only dynamically added properties are enumerated by `for-in` loops. Declared variables and methods of classes are not enumerated in `for-in` loops. This means that most classes in the ActionScript API do not display any properties in a `for-in` loop. The generic type Object is still a dynamic object and displays properties in a `for-in` loop.

To list all of the public properties and methods of a class or class instance, use the `describeType()` method and to parse the results use the E4X API. The `describeType()` method is in the flash.system package. The method's only parameter is the object that you want to introspect. You can pass any ActionScript value to it, including all available ActionScript types such as object instances, primitive types such as uint, and class objects. The return value of the `describeType()` method is an E4X XML object that contains an XML description of the object's type. For more information about using E4X, see the *Flex 2 Developer's Guide*.

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

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

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

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

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

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

```
      // List accessors as properties:
      for each (var a:XML in classInfo..accessor) {
        ta1.text += "Property " + a.@name + "=" + button1[a.@name] + " (" +
      a.@type +")\n";
      }

      // List the object's methods:
      for each (var m:XML in classInfo..method) {
        ta1.text += "Method " + m.@name + "():" + m.@returnType + "\n";
      }
    }
    ]]></mx:Script>
    <mx:Button label="Submit" id="button1"/>
    <mx:TextArea id="ta1" width="400" height="200"/>
    <mx:TextArea id="ta2" width="400" height="200"/>
</mx:Application>
```

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

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

Using the drag-and-drop feature

When you convert drag-and-drop code, be aware of the following changes:

- The `doDrag()` method takes an additional required attribute, `mouse_event`. This attribute is the MouseEvent object that contains the mouse information for the start of the drag.
- All drag-and-drop-specific events are now DragEvent class events.
- For a drop target to accept an item for dropping, it must call the `acceptDragDrop()` method, not use the `event.handled` property.

The following example lets you drag one of two colored canvases onto a larger canvas to apply the color to the larger canvas:

Flex 1.x:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
    import mx.managers.Dragmanager;
    function dragIt(event, text, format) {
      var ds:mx.core.DragSource = new mx.core.DragSource();
      ds.addData(text, format);
      DragManager.doDrag(event.target, ds, mx.containers.Canvas,
        {backgroundColor:event.target.getStyle('backgroundColor'),
        width:30, height:30});
    }
    function doDragEnter(event) {
      if (event.dragSource.hasFormat('color')) {
        event.handled = true;
      }
    }
    function doDragDrop(event) {
      var data = event.dragSource.dataForFormat('color');
      myCanvas.setStyle("backgroundColor", data);
    }
  ]]></mx:Script>
  <mx:HBox>
    <mx:Canvas backgroundColor="#FF0000" borderStyle="solid" width="30"
      height="30" mouseMove="dragIt(event, 'red', 'color')"/>
    <mx:Canvas backgroundColor="#00FF00" borderStyle="solid" width="30"
      height="30" mouseMove="dragIt(event, 'green', 'color')"/>
  </mx:HBox>
  <mx:Label text="Drag the item into this canvas"/>
  <mx:Canvas id="myCanvas" backgroundColor="#FFFFFF" borderStyle="solid"
    width="100" height="100" dragEnter="doDragEnter(event)"
    dragDrop="doDragDrop(event)"/>
 </mx:Application>
```

Flex 2:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
    import mx.managers.DragManager;
    // Import events package for MouseEvent:
    import mx.events.*;
    // Specify types for all arguments:
    function dragIt(event:MouseEvent, text:String, format:String) {
      var ds:mx.core.DragSource = new mx.core.DragSource();
      ds.addData(text, format);
      // New doDrag signature:
      DragManager.doDrag(event.target, ds, event);
    }
    function doDragEnter(event:DragEvent) {
      if (event.dragSource.hasFormat('color')) {
        /? For a drop target to accept an item for dropping, it calls the
        // acceptDragDrop() method and does not use the
        // event.handled property:
        DragManager.acceptDragDrop(event.target);
      }
    }
    function doDragDrop(event:DragEvent) {
      var data:Object = event.dragSource.dataForFormat('color');
      myCanvas.setStyle("backgroundColor", data);
    }
  ]]></mx:Script>
  <mx:HBox>
    <mx:Canvas backgroundColor="#FF0000" borderStyle="solid" width="30"
      height="30" mouseMove="dragIt(event, 'red', 'color')"/>
    <mx:Canvas backgroundColor="#00FF00" borderStyle="solid" width="30"
      height="30" mouseMove="dragIt(event, 'green', 'color')"/>
  </mx:HBox>
  <mx:Label text="Drag the item into this canvas"/>
  <mx:Canvas id="myCanvas" backgroundColor="#FFFFFF" borderStyle="solid"
    width="100" height="100" dragEnter="doDragEnter(event)"
    dragDrop="doDragDrop(event)"/>
 </mx:Application>
```

Other issues

In addition to the changes shown here, you might also encounter the issues described in this section when you convert drag-and-drop code.

Accessing event targets

Because a TextArea control uses a Flash TextField control, and mouse events are dispatched by Flash Player, not Flex, you must use the `currentTarget` property instead of the `target` property in the `doDrag()` method when you start a drag from a TextArea control.

Detecting keys

Because ActionScript no longer has a Key class, and the Keyboard class replaces a limited set of the Key class's functionality, three new Boolean properties, `ctrlKey`, `shiftKey`, and `AltKey`, were added to the DragEvent object. The new properties represent these keys' states when the user drags an item over a drop target. Typically, when a user holds the Shift, Control, or Alt key down during a drag-and-drop operation, the user wants to change the default behavior of the dragged item.

The following example checks if the user is pressing the Control key when dragging over the target (to copy data), and sets the action in the DragManager to reflect its state.

Flex 1.5:

```
private function doDragOver(event:Event) {
   event.target.showDropFeedback(event);
   if (Key.isDown(Key.CONTROL)) {
      ...
   } else if (Key.isDown(Key.SHIFT)) {
      ...
   }
}
```

Flex 2:

```
private function doDragOver(event:DragEvent) {
   event.target.showDropFeedback(event);
   if (event.ctrlKey) {
      ...
   } else if (event.shiftKey) {
      ...
   }
}
```

Controlling the feedback indicator

To control the feedback indicator that accompanies a drag proxy, you now use the new `showFeedback()` and `getFeedback()` methods of the DragManager class. The indicator shows what happens if you try to drop the item; for example, a red circle with a white x represents an aborted drop, or a green circle with a white plus (+) indicates a valid drop.

In Flex 1.x, you could change the feedback indicator with the `action` property of the event. In Flex 2, use the `showFeedback()` method to control the value of this property for all DragManager-related events. To *get* this value on any DragEvent object, you now use the `getFeedback()` method.

Setting actions

You no longer set the `action` property of the event object. Instead, you must call the `DragManager.setFeedback()` method, as the following example shows.

Flex 1.5:

```
private function doDragOver(event:Event):Void {
   // If the Control key is down, show the COPY drag feedback appearance.
   if (event.ctrlKey) {
     event.action = DragManager.COPY;
   } else {
     event.action = DragManager.MOVE;
   }
}
```

Flex 2:

```
private function doDragOver(event:DragEvent):void {
   // If the Control key is down, show the COPY drag feedback appearance.
   if (event.ctrlKey) {
     DragManager.showFeedback(DragManager.COPY);
   } else {
     DragManager.showFeedback(DragManager.MOVE);
   }
}
```

Using Timer

The `setInterval()` and `clearInterval()` methods were deprecated in favor of the Timer class. You can still use these methods; they are in the flash.util package.

When you use Timers, keep the following in mind:

- When a Timer is first created with the new operator, it is stopped; you must use the `start()` method to start it.
- Instances of the Timer class dispatch events that you handle like any other event.

The following example creates and destroys a Timer object each time you click the Start and Stop buttons. Setting the timer to `null` allows it to be garbage collected.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
    import flash.util.Timer;
    import flash.events.TimerEvent;

    private var timer:Timer;

    private function startTimer():void {
      timer = new Timer(1000);
      timer.addEventListener(TimerEvent.TIMER, timerHandler);
      timer.start();
    }
    private function stopTimer():void {
      timer.stop();
      timer = null;
    }
    private function timerHandler(event:TimerEvent):void {
      trace("timer");
    }
  ]]></mx:Script>
  <mx:Button label="Start Timer" click="startTimer();"/>
  <mx:Button label="Stop Timer" click="stopTimer();"/>
</mx:Application>
```

You can also use the `reset()` method rather than the `stop()` method to stop the timer. The difference between `stop()` and `reset()` is that `stop()` stops the timer but does not reset its count, while `reset()` both stops and resets.

Using the Preloader

The Application container supports an application preloader that uses a download progress bar to show the download progress of an application SWF file. By default, the application preloader is enabled. The preloader keeps track of how many bytes are downloaded and continually updates the progress bar.

By default, the application preloader uses the DownloadProgressBar class in the mx.preloaders package to display the download progress bar. To create a custom download progress bar, you can either create a subclass of the DownloadProgressBar class, or create a subclass of the flash.display.Sprite class that implements the mx.preloaders.IPreloaderDsiplay interface.

The operation of the download progress bar is defined by a set of events. These events are dispatched by the Preloader class. A custom download progress bar must handle these events.

For more information, see Chapter 14, "Using the Application Container," in the *Flex 2 Developer's Guide*.

Accessing request data

You can pass request data to any Flex application by using `flashVars` variables in the `<object>` and `<embed>` tags in the wrapper. If you are using Flex Data Services, you can specify the request data as query string parameters. The server converts these to `flashVars` variables when it generates the wrapper. The `flashVars` variables are a series of URL-encoded name and value pairs, as the following example shows:

```
flashVars='firstname=Nick&middlename=D&lastname=Danger'
```

The way in which you access these variables is changed. In Flex 1.x, you could access the values of `flashVars` variables by declaring a public global variable of the same name. Using the previous example, you could access the values of the `firstname`, `middlename`, and `lastname` variables in your Flex application by just declaring them, as the following example shows:

```
<mx:Script>
  var firstname; // Initialized to "Nick".
  var middlename; // Initialized to "D".
  var lastname; // Initialized to "Danger".
</mx:Script>
```

In Flex 2, you must use the `Application.application.parameters` property to get the values of these variables. The `parameters` property is an Object, which is a dynamic class that you can use to store name and value pairs pass in as `flashVars` variables. The following example sets variables by using the Flex 2 syntax:

```
<mx:Script>
  public var fName:String = Application.application.parameters.firstname;
  public var mName:String = Application.application.parameters.middlename;
  public var lName:String = Application.application.parameters.lastname;
</mx:Script>
```

For more information on using the Application.application.parameters property, see Chapter 34, "Communicating with the Wrapper," in the *Flex 2 Developer's Guide*.