Adobe® Flex™ 2

Programming ActionScript 3.0

Adobe®

Contents

PART 1

Overview of ActionScript Programming

1

This part describes fundamental programming concepts in ActionScript 3.0.

The following chapters are included:

About This Manual

This manual provides a foundation for developing applications in ActionScript 3.0. To best understand the ideas and techniques described, you should already be familiar with general programming concepts such as data types, variables, loops, and functions. You should also understand basic object-oriented programming concepts like classes and inheritance. Prior knowledge of ActionScript 1.0 or ActionScript 2.0 is helpful but not necessary.

Contents

Using this manual

This manual is divided into the following parts:

Part	Description
Part 1, "Overview of ActionScript Programming"	Discusses core ActionScript 3.0 concepts, including language syntax, statements and operators, the ECMAScript edition 4 draft language specification, object-oriented ActionScript programming, and the new approach to managing display objects on the Adobe® Flash® Player 9 display list.
Part 2, "Core ActionScript 3.0 Data Types and Classes"	Describes top-level data types in ActionScript 3.0 that are also part of the ECMAScript draft specification.
Part 3, "Flash Player APIs"	Describes important features that are implemented in packages and classes specific to Adobe Flash Player 9, including event handling, networking and communications, file input and output, the external interface, the application security model, and more.

The manual contains numerous sample files for important or commonly used classes to demonstrate application programming concepts for those APIs. Sample files are packaged in ways to make them easier to load and use with Adobe® Flex™ Builder 2 and may include wrapper files. However, the core sample code is pure ActionScript 3.0 that you can use in whichever development environment you prefer.

ActionScript 3.0 can be written and compiled a number of ways, including:

- Using the Adobe Flex Builder 2 development environment
- Using any text editor and a command-line compiler, such as the one provided with Flex Builder 2
- Using the Adobe® Flash® CS3 authoring tool from Adobe

For more information about ActionScript development environments, see Chapter 1, "Introduction to ActionScript 3.0."

To understand the code samples in this manual, you don't need to have prior experience using integrated development environments for ActionScript, such as Flex Builder or the Flash authoring tool. You will, however, want to refer to the documentation for those tools to learn how to use them to write and compile ActionScript 3.0 code. For more information, see "Accessing ActionScript documentation" on page 13.

Accessing ActionScript documentation

Because this manual focuses on describing ActionScript 3.0, which is a rich and powerful object-oriented programming language, it does not extensively cover the application development process or workflow within a particular tool or server architecture. So in addition to *Programming ActionScript 3.0*, you'll want to consult other sources of documentation as you design, develop, test, and deploy ActionScript 3.0 applications.

ActionScript 3.0 documentation

This manual familiarizes you with the concepts behind the ActionScript 3.0 programming language and gives you implementation details and samples illustrating important language features. However, this manual is not a complete language reference. For that, see the *ActionScript 3.0 Language Reference*, which describes every class, method, property, and event in the language. The *ActionScript 3.0 Language Reference* provides detailed reference information about the core language, Flex MXML classes and components (in the mx packages), and Flash Player APIs (in the flash packages).

Flex documentation

If you use the Flex development environment, you may want to consult these manuals:

Book	Description
Flex 2 Developer's Guide	Describes how to develop your dynamic web applications.
Getting Started with Flex 2	Contains an overview of Flex features and application development procedures.
Building and Deploying Flex 2 Applications	Describes how to build and deploy Flex applications.
Creating and Extending Flex 2 Components	Describes how to create and extend Flex components.
Migrating Applications to Flex 2	Provides an overview of the migration process, as well as detailed descriptions of changes in Flex and ActionScript.
Using Flex Builder 2	Contains comprehensive information about all Flex Builder features, for every level of Flex Builder users.
ActionScript 3.0 Language Reference	Provides descriptions, syntax, usage, and code examples for the Flex API.

Developer Center

The Adobe Developer Center is a your resource for up-to-the-minute information on ActionScript, articles about real-world application development, and information about important emerging issues. View the Developer Center at www.adobe.com/devnet/.

CHAPTER 1

Introduction to ActionScript 3.0

1

This chapter provides an overview of ActionScript 3.0, the newest and most revolutionary version of ActionScript.

Contents

About ActionScript

ActionScript is the programming language for the Flash Player run-time environment. It enables interactivity, data handling, and much more in Flash content and applications.

ActionScript is executed by the ActionScript Virtual Machine, which is part of Flash Player. ActionScript code is typically compiled into bytecode format by a compiler, such as the one built into the Flash authoring tool or Flex Builder, or that is available in the Flex SDK and the Flex Data Services. The bytecode is embedded in SWF files, which are executed by the Flash Player, the run-time environment.

ActionScript 3.0 offers a robust programming model that will be familiar to developers with a basic knowledge of object-oriented programming. Some of the key features of ActionScript 3.0 include the following:

- A new ActionScript Virtual Machine, called AVM2, that uses a new bytecode instruction set and provides significant performance improvements
- A more modern compiler code base that adheres much more closely to the ECMAScript (ECMA 262) standard and that performs deeper optimizations than previous versions of the compiler
- An expanded and improved application programming interface (API), with low-level control of objects and a true object-oriented model

- A core language based on the upcoming ECMAScript (ECMA-262) edition 4 draft language specification
- An XML API based on ECMAScript for XML (E4X), as specified in ECMA-357 edition 2 specification. E4X is a language extension to ECMAScript that adds XML as a native data type of the language.
- An event model based on the Document Object Model (DOM) Level 3 Events Specification

Advantages of ActionScript 3.0

ActionScript 3.0 goes beyond the scripting capabilities of previous versions of ActionScript. It is designed to facilitate the creation of highly complex applications with large data sets and object-oriented, reusable code bases. While ActionScript 3.0 is not required for content that runs in Adobe Flash Player 9, it opens the door to performance improvements that are only available with the AVM2, the new virtual machine. ActionScript 3.0 code can execute up to ten times faster than legacy ActionScript code.

The older version of ActionScript Virtual Machine, AVM1, executes ActionScript 1.0 and ActionScript 2.0 code. AVM1 is supported by Flash Player 9 for backward compatibility with existing and legacy content. For more information, see “Compatibility with previous versions” on page 20.

What’s new in ActionScript 3.0

Although ActionScript 3.0 contains many classes and features that will be familiar to ActionScript programmers, ActionScript 3.0 is architecturally and conceptually different from previous versions of ActionScript. The enhancements in ActionScript 3.0 include new features of the core language and an improved Flash Player API that provides increased control of low-level objects.

Core language features

The core language defines the basic building blocks of the programming language, such as statements, expressions, conditions, loops, and types. ActionScript 3.0 contains many new features that speed up the development process.

Run-time exceptions

ActionScript 3.0 reports more error conditions than previous versions of ActionScript. Run-time exceptions are used for common error conditions, improving the debugging experience and enabling you to develop applications that handle errors robustly. Run-time errors can provide stack traces annotated with source file and line number information, helping you quickly pinpoint errors.

Run-time types

In ActionScript 2.0, type annotations were primarily a developer aid; at run time, all values were dynamically typed. In ActionScript 3.0, type information is preserved at run time, and used for a number of purposes. Flash Player 9 performs run-time type checking, improving the system's type safety. Type information is also used to represent variables in native machine representations, improving performance and reducing memory usage.

Sealed classes

ActionScript 3.0 introduces the concept of sealed classes. A sealed class possesses only the fixed set of properties and methods that were defined at compile time; additional properties and methods cannot be added. This enables stricter compile-time checking, resulting in more robust programs. It also improves memory usage by not requiring an internal hash table for each object instance. Dynamic classes are also possible using the `dynamic` keyword. All classes in ActionScript 3.0 are sealed by default, but can be declared to be dynamic with the `dynamic` keyword.

Method closures

ActionScript 3.0 enables a method closure to automatically remember its original object instance. This feature is useful for event handling. In ActionScript 2.0, method closures would not remember what object instance they were extracted from, leading to unexpected behavior when the method closure was invoked. The mx.utils.Delegate class was a popular workaround, but it is no longer needed.

ECMAScript for XML (E4X)

ActionScript 3.0 implements ECMAScript for XML (E4X), recently standardized as ECMA-357. E4X offers a natural, fluent set of language constructs for manipulating XML. In contrast to traditional XML-parsing APIs, XML with E4X performs like a native data type of the language. E4X streamlines the development of applications that manipulate XML by drastically reducing the amount of code needed. For more information about the ActionScript 3.0 implementation of E4X, see Chapter 11, “Working with XML,” on page 311.

To view ECMA’s E4X specification, go to www.ecma-international.org/publications/files/ECMA-ST/ECMA-357.pdf .

Regular expressions

ActionScript 3.0 includes native support for regular expressions so that you can quickly search for and manipulate strings. ActionScript 3.0 implements support for regular expressions as they are defined in the ECMAScript edition 3 language specification (ECMA-262).

Namespaces

Namespaces are similar to the traditional access specifiers used to control visibility of declarations (`public`, `private`, `protected`). They work as custom access specifiers, which can have names of your choice. Namespaces are outfitted with a Universal Resource Identifier (URI) to avoid collisions, and are also used to represent XML namespaces when you work with E4X.

New primitive types

ActionScript 2.0 has a single numeric type, Number, a double-precision, floating point number. ActionScript 3.0 contains the int and uint types. The int type is a 32-bit signed integer that lets ActionScript code take advantage of the fast integer math capabilities of the CPU. The int type is useful for loop counters and variables where integers are used. The uint type is an unsigned, 32-bit integer type that is useful for RGB color values, byte counts, and more.

Flash Player API features

The Flash Player API in ActionScript 3.0 contains many new classes that allow you to control objects at a low level. The architecture of the language is completely new and more intuitive. While there are too many new classes to cover in detail here, the following sections highlight some significant changes.

DOM3 event model

Document Object Model Level 3 event model (DOM3) provides a standard way of generating and handling event messages so that objects within applications can interact and communicate, maintaining their state and responding to change. Patterned after the World Wide Web Consortium DOM Level 3 Events Specification, this model provides a clearer and more efficient mechanism than the event systems available in previous versions of ActionScript.

Events and error events are located in the flash.events package. The Flex application framework uses the same event model as the Flash Player API, so the event system is unified across the Flash platform.

Display list API

The API for accessing the Flash Player display list—the tree that contains any visual elements in a Flash application—consists of classes for working with visual primitives in Flash.

The new Sprite class is a lightweight building block, similar to MovieClip but more appropriate as a base class for UI components. The new Shape class represents raw vector shapes. These classes can be instantiated naturally with the `new` operator and can be dynamically re-parented at any time.

Depth management is now automatic and built into Flash Player, rendering assignment of depth numbers unnecessary. New methods are provided for specifying and managing the z-order of objects.

Handling dynamic data and content

ActionScript 3.0 contains mechanisms for loading and handling assets and data in your Flash application that are intuitive and consistent across the API. The new Loader class provides a single mechanism for loading SWF files and image assets and provides a way to access detailed information about loaded content. The URLLoader class provides a separate mechanism for loading text and binary data in data-driven applications. The Socket class provides a means to read and write binary data to server sockets in any format.

Low-level data access

Various APIs provide low-level access to data that was never before available in ActionScript. For data that is being downloaded, the URLStream class, which is implemented by URLLoader, provides access to data as raw binary data while it is being downloaded. The ByteArray class lets you optimize reading, writing, and working with binary data. The new Sound API provides detailed control of sound through the SoundChannel and SoundMixer classes. New APIs dealing with security provide information about the security privileges of a SWF or loaded content, enabling you to better handle security errors.

Working with text

ActionScript 3.0 contains a flash.text package for all text-related APIs. The TextLineMetrics class provides detailed metrics for a line of text within a text field; it replaces the `TextField.getLineMetrics()` method in ActionScript 2.0. The TextField class contains a number of interesting new low-level methods that can provide specific information about a line of text or a single character in a text field. These methods include `getCharBoundaries()`, which returns a rectangle representing the bounding box of a character, `getCharIndexAtPoint()`, which returns the index of the character at a specified point, and `getFirstCharInParagraph()`, which returns the index of the first character in a paragraph. Line-level methods include `getLineLength()`, which returns the number of characters in a specified line of text, and `getLineText()`, which returns the text of the specified line. A new Font class provides a means to manage embedded fonts in SWF files.

Compatibility with previous versions

As always, Flash Player provides full backward compatibility with previously published content. Any content that ran in previous versions of Flash Player runs in Flash Player 9. The introduction of ActionScript 3.0 in Flash Player 9, however, does present some challenges for interoperability between old and new content running in Flash Player 9. The compatibility issues include the following:

- A single SWF file cannot combine ActionScript 1.0 or 2.0 code with ActionScript 3.0 code.
- ActionScript 3.0 code can load a SWF file written in ActionScript 1.0 or 2.0, but it cannot access the SWF file's variables and functions.
- SWF files written in ActionScript 1.0 or 2.0 cannot load SWF files written in ActionScript 3.0. This means that SWF files authored in Flash 8 or Flex Builder 1.5 or earlier versions cannot load ActionScript 3.0 SWF files.

The only exception to this rule is that an ActionScript 2.0 SWF file can replace itself with an ActionScript 3.0 SWF file, as long as the ActionScript 2.0 SWF file hasn't previously loaded anything into any of its levels. An ActionScript 2.0 SWF file can do this through a call to `loadMovieNum()`, passing a value of 0 to the `level` parameter.

- In general, SWF files written in ActionScript 1.0 or 2.0 must be migrated if they are to work together with SWF files written in ActionScript 3.0. For example, say you created a media player using ActionScript 2.0. The media player loads various content that was also created using ActionScript 2.0. You cannot create new content in ActionScript 3.0 and load it in the media player. You must migrate the video player to ActionScript 3.0.

  If, however, you create a media player in ActionScript 3.0, that media player can perform simple loads of your ActionScript 2.0 content.

The following diagram summarizes the limitations of previous versions of Flash Player in relation to loading new content and executing code, as well as the limitations for cross-scripting between SWF files written in different versions of ActionScript.

Supported functionality	**Run-time environment**		
	Flash Player 7	Flash Player 8	Flash Player 9
Can load SWFs published for	7 and earlier	8 and earlier	9 and earlier
Contains this AVM	AVM1	AVM1	AVM1 and AVM2
Runs SWFs written in ActionScript	1.0 and 2.0	1.0 and 2.0	1.0 and 2.0, and 3.0

Supported functionality*	**Content created in**	
	ActionScript 1.0 and 2.0	ActionScript 3.0
Can load content and execute code in content created in	ActionScript 1.0 and 2.0 only	ActionScript 1.0 and 2.0, and ActionScript 3.0
Can cross script content created in	ActionScript 1.0 and 2.0 only†	ActionScript 3.0‡

* Content running in Flash Player 9 or later. Content running in Flash Player 8 or earlier can load, display, execute, and cross-script only ActionScript 1.0 and 2.0.

† ActionScript 3.0 through Local Connection.

‡ ActionScript 1.0 and 2.0 through LocalConnection.

CHAPTER 2

Getting Started with ActionScript

2

This chapter provides a no-frills, step-by-step approach to building a simple ActionScript 3.0 application.

The ActionScript 3.0 programming language can be used from within a number of different development environments, including Macromedia Flash from Adobe and Adobe Flex Builder 2. This chapter will show how to create modular ActionScript code that can be used from within either of these application development environments.

Contents

The basic ActionScript development process

No matter whether your ActionScript project is large or small, using a process to design and develop your application will help you work more efficiently and effectively. Here are a set of steps for a basic development process for building an application that uses ActionScript 3.0:

1. Design your application.

   You should describe your application in some way before you start building it.

2. Compose your ActionScript 3.0 code.

   You can create ActionScript code using Flash, Flex Builder, Macromedia Dreamweaver® from Adobe, or a text editor.

3. Create a Flash or Flex application file to run your code.

   In the Flash authoring tool, this involves creating a new FLA file, setting up the publish settings, adding user interface components to the application, and referencing the ActionScript code. In the Flex development environment, creating a new application file involves defining the application and adding user interface components using MXML, and referencing the ActionScript code.
4. Publish and test your ActionScript application.

   This involves running your application from within the Flash authoring or Flex development environment, and making sure it does everything you intended.

Note that you don't necessarily have to follow these steps in order, or completely finish one step before working on another. For example, you might design one screen of your application (step 1), then create the graphics, buttons, and so forth (step 3), before writing ActionScript code (step 2) and testing (step 4). Or you might design part of it, then add one button or interface element at a time, writing ActionScript for each one and testing it as it's built. So while it's convenient to remember these four stages of the development process, in real-world development it's usually more effective to move back and forth among the stages as appropriate.

Options for organizing your code

You can use ActionScript 3.0 code to power everything from simple graphics animations to complex, client-server transaction processing systems. Depending on the type of application you're building, you may prefer to use one or more of these different ways of including ActionScript in your project.

Storing code in frames in a Flash timeline

In the Flash authoring environment, you can add ActionScript code to any frame in a timeline. This code will be executed while the movie is playing back, when the playhead enters that frame.

Placing ActionScript code in frames provides a simple way to add behaviors to applications built in the Flash authoring tool. You can add code to any frame in the main timeline or to any frame in the timeline of any MovieClip symbol. However, this flexibility comes with a cost. When you build larger applications, it becomes easy to lose track of which frames contain which scripts, which can make the application more difficult to maintain over time.

Many developers simplify the organization of their ActionScript code in the Flash authoring tool by placing code only in the first frame of a timeline, or on a specific layer in the Flash document. This makes it easier to locate and maintain the code in your Flash FLA files. However, in order to use the same code in another Flash or Flex project, you must copy and paste the code into the new file.

If you want to be able to use your ActionScript code in other Flash or Flex projects in the future, you will want to store your code in external ActionScript files (text files with the .as extension).

Embedding code in Flex MXML files

In a Flex development environment, you can include ActionScript code inside an `<mx:Script>` tag in a Flex MXML file. Inline ActionScript code like this has the same drawback as code placed on a frame in Flash: you cannot reuse the code without cutting and pasting it to a new source location.

NOTE

You can specify a source parameter for an `<mx:Script>` tag, which lets you insert ActionScript code as if it was typed directly within the `<mx:Script>` tag. However, the source file that you use cannot define its own class, which limits its reusability.

Storing code in ActionScript files

If your project involves significant ActionScript code, the best way to organize your code is in separate ActionScript source files (text files with the .as extension). An ActionScript file can be structured in one of two ways, depending on how you intend to use it in your application:

- Unstructured ActionScript code: lines of ActionScript code, including statements or function definitions, written as though they were entered directly in a timeline script, MXML file, etc.

  ActionScript written in this way can be accessed using the `include` statement in ActionScript, or the `<mx:Script>` tag in Flex MXML. The ActionScript `include` statement causes the contents of an external ActionScript file to be inserted at a specific location and within a given scope in a script, as if it were entered there directly. In the Flex MXML language, the `<mx:Script>` tag lets you specify a source attribute that identifies an external ActionScript file to be loaded at that point in the application. For example, the following tag will load an external ActionScript file named Box.as:

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

- ActionScript class definition: a definition of an ActionScript class, including its method and property definitions.

  When you define a class, you can access the ActionScript code in the class by creating an instance of the class and using its properties, methods, and events, just as you would with any of the built-in ActionScript classes. This requires two parts:

  - Use the `import` statement to specify the full name of the class, so the ActionScript compiler knows where to find it. For example, if you want to use the MovieClip class in ActionScript, you first need to import that class using its full name, including package and class:

    ```
    import flash.display.MovieClip;
    ```

    Alternatively, you can import the package that contains the MovieClip class, which is equivalent to writing separate import statements for each class in the package:

    ```
    import flash.display.*;
    ```

    The only exceptions to the rule that a class must be imported in order to refer to that class in your code are the top-level classes which are not defined in a package.
  - Write code which specifically refers to the class name (usually declaring a variable with that class as its data type, and creating an instance of the class to store in the variable). By referring to another class name in ActionScript code, you tell the compiler to load the definition of that class. For example, given an external class called Box, this statement causes a new instance of the Box class to be created:

    ```
    var smallBox:Box = new Box(10,20);
    ```

    When the compiler comes across the reference to the Box class for the first time, it searches the loaded source code to locate the Box class definition.

Example: Creating a basic application

You can create external ActionScript source files with an .as extension using Flash, Flex Builder, Dreamweaver, or any text editor.

ActionScript 3.0 can be used within a number of application development environments, including the Flash authoring and Flex Builder tools.

This section walks through the steps in creating and enhancing a simple ActionScript 3.0 application using the Flash authoring tool or the Flex Builder 2 tool. The application you'll build presents a simple pattern for using external ActionScript 3.0 class files in Flash and Flex applications. That pattern will apply to all of the other example applications in this book.

Designing your ActionScript application

You should have some idea about the application you want to build before you start building it.

The representation of your design can be as simple as the name of the application and a brief statement of its purpose, or as complicated as a set of requirements documents containing numerous Unified Modeling Language (UML) diagrams. This book doesn't discuss the discipline of software design in detail, but it's important to keep in mind that application design is an essential step in the development of ActionScript applications.

Our first example of an ActionScript application will be a standard "Hello World" application, so its design is very simple:

- The application will be called HelloWorld.
- It will display a single text field containing the words "Hello World!"
- In order to be easily reused, it will use a single object-oriented class, named Greeter, which can be used from within a Flash document or a Flex application.
- After you create a basic version of the application, you will add new functionality to have the user enter a user name and have the application check the name against a list of known users.

With that concise definition in place, you can start building the application itself.

Creating the HelloWorld project and the Greeter class

The design statement for the Hello World application said that its code should be easy to reuse. With this goal in mind, the application uses a single object-oriented class, named Greeter, which is used from within an application that you create in Flex Builder or the Flash authoring tool.

To create the HelloWorld project and Greeter class in Flex Builder:

1. In Flex Builder, select File > New> Flex Project,
2. If the New Flex Project dialog box asks you to select a Flex Server Technology, select Basic, and then click Next.
3. Type HelloWorld as the Project Name, and then click Finish.

   Your new project will be created and should be showing in the Navigator panel. By default the project should already contain a file named HelloWorld.mxml, and that file should be open in the Editor panel.
4. Now to create a custom ActionScript class file in the Flex Builder tool, select File > New > ActionScript File.

5. In the New ActionScript File dialog box, select HelloWorld as the parent folder, type **Greeter.as** the filename, and then click Finish.

   A new ActionScript editing window is displayed.

   Continue with "Adding code to the Greeter class" on page 28.

Adding code to the Greeter class

The Greeter class defines an object, `Greeter`, that you will be able to use in your HelloWorld application.

To add code to the Greeter class:

1. Type the following code into the new file:

```
package
{
    public class Greeter
    {
        public function sayHello():String
        {
            var greeting:String;
            greeting = "Hello World!";
            return greeting;
        }
    }
}
```

   The Greeter class includes a single `sayHello() method`, which returns a string that says "Hello" to the user name that is given.

2. Select File > Save to save this ActionScript file.

The Greeter class is now ready to be used in a Flash or Flex application.

Creating an application that uses your ActionScript code

The Greeter class that you have built defines a self-contained set of software functions, but it does not represent a complete application. To use the class, you need to create a Flash document or Flex application.

The HelloWorld application creates an new instance of the Greeter class. Here's how to attach the Greeter class to your application.

To create an ActionScript application using Flex Builder:

1. Open the HelloWorld.mxml file, and type the following code:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
     layout="vertical"
     creationComplete = "initApp()" >

     <mx:Script>
       <![CDATA[
         private var myGreeter:Greeter = new Greeter();

         public function initApp():void
         {
           // says hello at the start, and asks for the user's name
           mainTxt.text = myGreeter.sayHello();
         }
       ]]>
     </mx:Script>

     <mx:TextArea id = "mainTxt" width="400" />

   </mx:Application>
   ```

   This Flex project includes three MXML tags:

   - An `<mx:Application>` tag, which defines the Application container
   - An `<mx:Script>` tag that includes some ActionScript code
   - An `<mx:TextArea>` tag, which defines a field to display text messages to the user

   The code in the `<mx:Script>` tag defines a `initApp()` method that is called when the application loads. The `initApp()` method sets the text value of the `mainTxt` TextArea to the "Hello World!" string returned by the `sayHello()` method of the custom class Greeter, which you just wrote.

2. Select File > Save to save the application.

Continue with "Publishing and testing your ActionScript application" on page 30.

Publishing and testing your ActionScript application

Software development is an iterative process. You write some code, try to compile it, and edit the code until it compiles cleanly. You run the compiled application, test it to see if it fulfills the intended design, and if it doesn't, you edit the code again until it does. The Flash and Flex Builder development environments offer a number of ways to publish, test, and debug your applications.

Here are the basic steps for testing the HelloWorld application in each environment.

To publish and test an ActionScript application using Flex Builder:

1. Select Run > Run. Make sure that the Project field shows "HelloWorld" and the Application file field shows "HelloWorld.mxml".
2. In the Run dialog box, click Run.

   The HelloWorld application starts.

   - If any errors or warnings are displayed in the Output window when you test your application, fix the causes of these errors in the HelloWorld.mxml or Greeter.as files, and then try testing the application again.
   - If there are no compilation errors, a browser window opens showing the Hello World application. The text "Hello World!" should be displayed.

You have just created a simple but complete object-oriented application that uses ActionScript 3.0. Continue with "Enhancing the HelloWorld application" on page 30.

Enhancing the HelloWorld application

To make the application a little more interesting, you'll now make it ask for and validate a user name against a predefined list of names.

First, you will update the Greeter class to add new functionality. Then you will update the Flex or Flash application to use the new functionality.

To update the Greeter.as file:

1. Open the Greeter.as file.

2. Change the contents of the file to the following (new and changed lines are shown in boldface):

```
package
{
  public class Greeter
  {
    /**
     * Defines the names that should receive a proper greeting.
     */
    public static var validNames:Array = ["Sammy", "Frank", "Dean"];

    /**
     * Builds a greeting string using the given name.
     */
    public function sayHello(userName:String = ""):String
    {
      var greeting:String;
      if (userName == "")
      {
        greeting = "Hello. Please type your user name, and then press
  the Enter key.";
      }
      else if (validName(userName))
      {
        greeting = "Hello, " + userName + ".";
      }
      else
      {
        greeting = "Sorry, " + userName + ", you are not on the list.";
      }
      return greeting;
    }

    /**
     * Checks whether a name is in the validNames list.
     */
    public static function validName(inputName:String = ""):Boolean
    {
      if (validNames.indexOf(inputName) > -1)
      {
        return true;
      }
      else
      {
        return false;
      }
    }
  }
}
```

The Greeter class now has a number of new features:

- The `validNames` array lists valid user names. The array is initialized to a list of three names when the Greeter class is loaded.
- The `sayHello()` method now accepts a user name and changes the greeting based on some conditions. If the `userName` is an empty string (`""`), the `greeting` property is set to prompt the user for a name. If the user name is valid, the greeting becomes `"Hello, *userName*.`" Finally, if either of those two conditions are not met, the `greeting` variable is set to `"Sorry, *userName*, you are not on the list."`
- The `validName()` method returns `true` if the `inputName` is found in the `validNames` array, and `false` if it is not found. The statement `validNames.indexOf(inputName)` checks each of the strings in the `validNames` array against the `inputName` string. The `Array.indexOf()` method returns the index position of the first instance of an object in an array, or the value -1 if the object is not found in the array.

Next you will edit the Flash or Flex file that references this ActionScript class.

To modify the application using Flex Builder:

1. Open the HelloWorld.mxml file.
2. Next modify the `<mx:TextArea>` tag to indicate to the user that is is for display only, by changing the background color to a light gray and preventing the user from editing the displayed text:

```
<mx:TextArea id = "mainTxt" width="400" backgroundColor="#DDDDDD"
editable="false" />
```

3. Now add the following lines right after the `<mx:TextArea>` closing tag. These lines create a new TextInput field that lets the user enter a user name value:

```
<mx:HBox width="400">
  <mx:Label text="User Name:"/>
  <mx:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
</mx:HBox>
```

The `enter` attribute specifies that when the user presses the Enter key in the `userNameTxt` field, the text in the field will be passed to the `Greeter.sayHello()` method, and the greeting displayed in the `mainTxt` field will change accordingly.

The final contents of the HelloWorld.mxml file should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
  layout="vertical"
  creationComplete = "initApp()" >

  <mx:Script>
    <![CDATA[
          private var myGreeter:Greeter = new Greeter();

          public function initApp():void
          {
         // says hello at the start, and asks for the user's name
         mainTxt.text = myGreeter.sayHello();
          }

      ]]>
  </mx:Script>

  <mx:TextArea id = "mainTxt" width="400" backgroundColor="#DDDDDD"
  editable="false" />

  <mx:HBox width="400">
      <mx:Label text="User Name:"/>
      <mx:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
  myGreeter.sayHello(userNameTxt.text);" />
  </mx:HBox>

</mx:Application>
```

4. Save the edited HelloWorld.mxml file. Select File > Run to run the application.

   When you run the application, you will be prompted to enter a user name. If it is valid (Sammy, Frank, or Dean), the application will display the "`Hello, *userName*`" confirmation message.

Running subsequent examples

Having developed and run the "Hello World" ActionScript 3.0 application, you should have the basic knowledge you need to run the other code examples presented in this book.

Subsequent code examples will not be presented in a step-by-step tutorial format as this one was. The relevant ActionScript 3.0 code in each example will be highlighted and discussed, but instructions about running the examples in specific development environments won't be provided. However the example files distributed with this book will include all of the files you need to compile and run the examples easily in your chosen development environment.

CHAPTER 3

ActionScript Language and Syntax

3

ActionScript 3.0 comprises both the core ActionScript language and the Flash Player Application Programming Interface (API). The core language is the part of ActionScript that implements the draft ECMAScript (ECMA-262), Edition 4 draft language specification. The Flash Player API provides programmatic access to Flash Player.

This chapter provides a brief introduction to the core ActionScript language and syntax. After reading this chapter, you should have a basic understanding of how to work with data types and variables, how to use proper syntax, and how to control the flow of data in your program.

Contents

Language overview

Objects lie at the heart of the ActionScript 3.0 language—they are its fundamental building blocks. Every variable you declare, every function you write, and every class instance you create is an object. You can think of an ActionScript 3.0 program as a group of objects that carry out tasks, respond to events, and communicate with one another.

Programmers familiar with object-oriented programming (OOP) in Java or C++ may think of objects as modules that contain two kinds of members: data stored in member variables or properties, and behavior accessible through methods. The ECMAScript edition 4 draft, the standard upon which ActionScript 3.0 is based, defines objects in a similar but slightly different way. In the ECMAScript draft, objects are simply collections of properties. These properties are containers that can hold not only data, but also functions or other objects. If a function is attached to an object in this way, it is called a method.

While the ECMAScript draft definition may seem a little odd to programmers with a Java or C++ background, in practice, defining object types with ActionScript 3.0 classes is very similar to the way classes are defined in Java or C++. The distinction between the two definitions of object is important when discussing the ActionScript object model and other advanced topics, but in most other situations the term *properties* means class member variables as opposed to methods. The *ActionScript 3.0 Language Reference*, for example, uses the term *properties* to mean variables or getter-setter properties. It uses the term *methods* to mean functions that are part of a class.

One subtle difference between classes in ActionScript and classes in Java or C++ is that in ActionScript, classes are not just abstract entities. ActionScript classes are represented by *class objects* that store the class's properties and methods. This allows for techniques that may seem alien to Java and C++ programmers, such as including statements or executable code at the top level of a class or package.

Another difference between ActionScript classes and Java or C++ classes is that every ActionScript class has something called a *prototype object*. In previous versions of ActionScript, prototype objects, linked together into *prototype chains,* served collectively as the foundation of the entire class inheritance hierarchy. In ActionScript 3.0, however, prototype objects play only a small role in the inheritance system. The prototype object can still be useful, however, as an alternative to static properties and methods if you want to share a property and its value among all the instances of a class.

In the past, advanced ActionScript programmers could directly manipulate the prototype chain with special built-in language elements. Now that the language provides a more mature implementation of a class-based programming interface, many of these special language elements, such as `__proto__` and `__resolve`, are no longer part of the language. Moreover, optimizations of the internal inheritance mechanism that provide significant Flash Player performance improvements preclude direct access to the inheritance mechanism.

Objects and classes

In ActionScript 3.0, every object is defined by a class. A class can be thought of as a template or a blueprint for a type of object. Class definitions can include variables and constants, which hold data values, and methods, which are functions that encapsulate behavior bound to the class. The values stored in properties can be *primitive values* or other objects. Primitive values are numbers, strings, or Boolean values.

ActionScript contains a number of built-in classes that are part of the core language. Some of these built-in classes, such as Number, Boolean and String, represent the primitive values available in ActionScript. Others, such as the Array, Math, and XML classes, define more complex objects that are part of the ECMAScript standard.

All classes, whether built-in or user-defined, derive from the Object class. For programmers with previous ActionScript experience, it is important to note that the Object data type is no longer the default data type, even though all other classes still derive from it. In ActionScript 2.0, the following two lines of code were equivalent because the lack of a type annotation meant that a variable would be of type Object:

```
var someObj:Object;
var someObj;
```

ActionScript 3.0, however, introduces the concept of untyped variables, which can be designated in the following two ways:

```
var someObj:*;
var someObj;
```

An untyped variable is not the same as a variable of type Object. The key difference is that untyped variables can hold the special value `undefined`, while a variable of type Object cannot hold that value.

You can define your own classes using the `class` keyword. You can declare class properties in three ways: constants can be defined with the `const` keyword, variables are defined with the `var` keyword, and getter and setter properties are defined by using the `get` and `set` attributes in a method declaration. You can declare methods with the `function` keyword.

You create an instance of a class by using the `new` operator. The following example creates an instance of the Date class called `myBirthday`.

```
var myBirthday:Date = new Date();
```

Packages and namespaces

Packages and namespaces are related concepts. Packages allow you to bundle class definitions together in a way that facilitates code sharing and minimizes naming conflicts. Namespaces allow you to control the visibility of identifiers, such as property and method names, and can be applied to code whether it resides inside or outside a package. Packages let you organize your class files, and namespaces let you manage the visibility of individual properties and methods.

Packages

Packages in ActionScript 3.0 are implemented with namespaces, but are not synonymous with them. When you declare a package, you are implicitly creating a special type of namespace that is guaranteed to be known at compile time. Namespaces, when created explicitly, are not necessarily known at compile time.

The following example uses the `package` directive to create a simple package containing one class:

```
package samples
{
  public class SampleCode
  {
    public var sampleGreeting:String;
    public function sampleFunction()
    {
      trace(sampleGreeting + " from sampleFunction()");
    }
  }
}
```

The name of the class in this example is SampleCode. Because the class is inside the samples package, the compiler automatically qualifies the class name at compile time into its fully qualified name: samples.SampleCode. The compiler also qualifies the names of any properties or methods, so that `sampleGreeting` and `sampleFunction()` become `samples.SampleCode.sampleGreeting` and `samples.SampleCode.sampleFunction()`, respectively.

Many developers, especially those with Java programming backgrounds, may choose to place only classes at the top level of a package. ActionScript 3.0, however, supports not only classes at the top level of a package, but also variables, functions, and even statements. One advanced use of this feature is to define a namespace at the top level of a package so that it will be available to all classes in that package. Note, however, that only two access specifiers, `public` and `internal`, are allowed at the top level of a package. Unlike Java, which allows you to declare nested classes private, ActionScript 3.0 supports neither nested nor private classes.

In many other ways, however, ActionScript 3.0 packages are similar to packages in the Java programming language. As you can see in the previous example, fully qualified package references are expressed using the dot operator (`.`), just as they are in Java. You can use packages to organize your code into an intuitive hierarchical structure for use by other programmers. This facilitates code sharing by allowing you to create your own package to share with others, and to use packages created by others in your code.

The use of packages also helps to ensure that the identifier names that you use are unique and do not conflict with other identifier names. In fact, some would argue that this is the primary benefit of packages. For example, two programmers who wish to share their code with each other may have each created a class called SampleCode. Without packages, this would create a name conflict, and the only resolution would be to rename one of the classes. With packages, however, the name conflict is easily avoided by placing one, or preferably both, of the classes in packages with unique names.

You can also include embedded dots in your package name to create nested packages. This allows you to create a hierarchical organization of packages. A good example of this is the flash.xml package provided by the Flash Player API. The flash.xml package is nested inside the flash package.

The flash.xml package contains the legacy XML parser that was used in previous versions of ActionScript. One reason that it now resides in the flash.xml package is that the name of the legacy XML class conflicts with the name of the new XML class that implements the XML for ECMAScript (E4X) specification functionality available in ActionScript 3.0.

Although moving the legacy XML class into a package is a good first step, most users of the legacy XML classes will import the flash.xml package, which will generate the same name conflict unless you remember to always use the fully qualified name of the legacy XML class (flash.xml.XML). To avoid this situation, the legacy XML class is now named XMLDocument, as the following example shows:

```
package flash.xml
{
  class XMLDocument {}
  class XMLNode {}
  class XMLSocket {}
}
```

Most of the Flash Player API is organized under the flash package. For example, the flash.display package contains the display list API, and the flash.events package contains the new event model. A detailed discussion of the Flash Player API packages can be found in Part 3 of this book. For more information, see "Flash Player APIs" on page 335.

Creating packages

ActionScript 3.0 provides significant flexibility in the way you organize your packages, classes, and source files. Previous versions of ActionScript allowed only one class per source file and required that the name of the source file match the name of the class. ActionScript 3.0 allows you to include multiple classes in one source file, but only one class in each file can be made available to code that is external to that file. In other words, only one class in each file can be declared inside a package declaration. You must declare any additional classes outside your package definition, which makes those classes invisible to code outside that source file. The name of the class declared inside the package definition must match the name of the source file.

ActionScript 3.0 also provides more flexibility in the way you declare packages. In previous versions of ActionScript, packages merely represented directories in which you placed source files, and you didn't declare packages with the `package` statement, but rather included the package name as part of the fully qualified class name in your class declaration. Although packages still represent directories in ActionScript 3.0, packages can contain more than just classes. In ActionScript 3.0, you use the `package` statement to declare a package, which means that you can also declare variables, functions, and namespaces at the top level of a package. You can even include executable statements at the top level of a package. If you do declare variables, functions, or namespaces at the top level of a package, the only attributes available at that level are `public` and `internal`, and only one package-level declaration per file can use the `public` attribute, whether that declaration is a class, variable, function, or namespace.

Packages are useful for organizing your code and for preventing name conflicts. You should not confuse the concept of packages with the unrelated concept of class inheritance. Two classes that reside in the same package will have a namespace in common, but are not necessarily related to each other in any other way. Likewise, a nested package may have no semantic relationship to its parent package.

Importing packages

If you want to use a class that is inside a package, you must import either the package or the specific class. This differs from ActionScript 2.0, where importing classes was optional.

For example, consider the SampleCode class example from earlier in this chapter. If the class resides in a package named samples, you must use one of the following import statements before using the SampleCode class:

```
import samples.*;
```

or

```
import samples.SampleCode;
```

In general, `import` statements should be as specific as possible. If you plan to use only the SampleCode class from the samples package, you should import only the SampleCode class rather than the entire package to which it belongs. Importing entire packages may lead to unexpected name conflicts.

You must also place the source code that defines the package or class within your *classpath*. The classpath is a user-defined list of local directory paths that determines where the compiler will search for imported packages and classes. The classpath is sometimes called the *build path* or *source path*.

After you have properly imported the class or package, you can use either the fully qualified name of the class (samples.SampleCode) or merely the class name by itself (SampleCode).

Fully qualified names are useful when identically named classes, methods, or properties result in ambiguous code, but can be difficult to manage if used for all identifiers. For example, the use of the fully qualified name results in verbose code when you instantiate a SampleCode class instance:

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

As the levels of nested packages increase, the readability of your code decreases. In situations where you are confident that ambiguous identifiers will not be a problem, you can make your code easier to read by using simple identifiers. For example, instantiating a new instance of the SampleCode class is much less verbose if you use only the class identifier:

```
var mySample:SampleCode = new SampleCode();
```

If you attempt to use identifier names without first importing the appropriate package or class, the compiler will not be able to find the class definitions. On the other hand, if you do import a package or class, any attempt to define a name that conflicts with an imported name will generate an error.

When a package is created, the default access specifier for all members of that package is `internal`, which means that, by default, package members are only visible to other members of that package. If you want a class to be available to code outside the package, you must declare that class to be `public`. For example, the following package contains two classes, SampleCode and CodeFormatter:

```
// SampleCode.as file
package samples
{
  public class SampleCode {}
}

// CodeFormatter.as file
package samples
{
  class CodeFormatter {}
}
```

The SampleCode class is visible outside the package because it is declared as a `public` class. The CodeFormatter class, however, is visible only within the samples package itself. If you attempt to access the CodeFormatter class outside the samples package, you will generate an error, as the following example shows:

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // okay, public class
var myFormatter:CodeFormatter = new CodeFormatter(); // error
```

If you want both classes to be available outside the package, you must declare both classes to be `public`. You cannot apply the `public` attribute to the package declaration.

Fully qualified names are useful for resolving name conflicts that may occur when using packages. Such a scenario may arise if you import two packages that define classes with the same identifier. For example, consider the following package, which also has a class named SampleCode:

```
package langref.samples
{
  public class SampleCode {}
}
```

If you import both classes, as follows, you will have a name conflict when referring to the SampleCode class:

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // name conflict
```

The compiler has no way of knowing which SampleCode class to use. To resolve this conflict, you must use the fully qualified name of each class, as follows:

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

NOTE

Programmers with a C++ background often confuse the `import` statement with `#include`. The `#include` directive is necessary in C++ because C++ compilers process one file at a time, and will not look in other files for class definitions unless a header file is explicitly included. ActionScript 3.0 has an `include` directive, but it is not designed to import classes and packages. To import classes or packages in ActionScript 3.0, you must use the `import` statement and place the source file that contains the package in the class path.

Namespaces

Namespaces give you control over the visibility of the properties and methods that you create. Think of the `public`, `private`, `protected`, and `internal` access control specifiers as built-in namespaces. If these predefined access control specifiers do not suit your needs, you can create your own namespaces.

If you are familiar with XML namespaces, much of this discussion will not be new to you, although the syntax and details of the ActionScript implementation are slightly different from those of XML. If you have never worked with namespaces before, the concept itself is straightforward, but the implementation has specific terminology that you will need to learn.

To understand how namespaces work, it helps to know that the name of a property or method always contains two parts: an identifier and a namespace. The identifier is what you generally think of as a name. For example, the identifiers in the following class definition are `sampleGreeting` and `sampleFunction()`:

```
class SampleCode
{
  var sampleGreeting:String;
  function sampleFunction () {
    trace(sampleGreeting + " from sampleFunction()");
  }
}
```

Whenever definitions are not preceded by a namespace attribute, their names are qualified by the default `internal` namespace, which means they are visible only to callers in the same package. If the compiler is set to strict mode, the compiler issues a warning that the `internal` namespace applies to any identifier without a namespace attribute. To ensure that an identifier is available everywhere, you must specifically precede the identifier name with the `public` attribute. In the previous example code, both `sampleGreeting` and `sampleFunction()` have a namespace value of `internal`.

There are three basic steps to follow when using namespaces. First, you must define the namespace using the `namespace` keyword. For example, the following code defines the `version1` namespace:

```
namespace version1;
```

Second, you apply your namespace by using it instead of an access control specifier in a property or method declaration. The following example places a function named `myFunction()` into the `version1` namespace:

```
version1 function myFunction() {}
```

Third, once you've applied the namespace, you can reference it with the `use` directive or by qualifying the name of an identifier with a namespace. The following example references the `myFunction()` function through the `use` directive:

```
use namespace version1;
myFunction();
```

You can also use a qualified name to reference the `myFunction()` function, as the following example shows:

```
version1::myFunction();
```

Defining namespaces

Namespaces contain one value, the Uniform Resource Identifier (URI), which is sometimes called the namespace name. A URI allows you to ensure that your namespace definition is unique.

You create a namespace by declaring a namespace definition in one of two ways. You can either define a namespace with an explicit URI, as you would define an XML namespace, or you can omit the URI. The following example shows how a namespace can be defined using a URI:

```
namespace flash_proxy = “http://www.adobe.com/flash/proxy”;
```

The URI serves as a unique identification string for that namespace. If you omit the URI, as in the following example, the compiler will create an unique internal identification string in place of the URI. You do not have access to this internal identification string.

```
namespace flash_proxy;
```

Once you define a namespace, with or without a URI, that namespace cannot be redefined in the same scope. An attempt to define a namespace that has been defined earlier in the same scope results in a compiler error.

If a namespace is defined within a package or a class, the namespace may not be visible to code outside that package or class unless the appropriate access control specifier is used. For example, the following code shows the `flash_proxy` namespace defined within the flash.utils package. In the following example, the lack of an access control specifier means that the `flash_proxy` namespace would be visible only to code within the flash.utils package and would not be visible to any code outside the package:

```
package flash.utils
{
    namespace flash_proxy;
}
```

The following code uses the `public` attribute to make the `flash_proxy` namespace visible to code outside the package:

```
package flash.utils
{
    public namespace flash_proxy;
}
```

Applying namespaces

Applying a namespace means placing a definition into a namespace. Definitions that can be placed into namespaces include functions, variables, and constants (you cannot place a class into a custom namespace).

Consider, for example, a function declared using the `public` access control namespace. Using the `public` attribute in a function definition places the function into the public namespace, which makes the function available to all code. Once you have defined a namespace, you can use the namespace that you defined the same way you would use the `public` attribute, and the definition will be available to code that can reference your custom namespace. For example, if you define a namespace `example1`, you can add a method called `myFunction()` using `example1` as an attribute, as the following example shows:

```
namespace example1;
class someClass
{
    example1 myFunction() {}
}
```

Declaring the `myFunction()` method using the namespace `example1` as an attribute means that the method belongs to the `example1` namespace.

You should bear in mind the following when applying namespaces:

- You can apply only one namespace to each declaration.
- There is no way to apply a namespace attribute to more than one definition at a time. In other words, if you want to apply your namespace to ten different functions, you must add your namespace as an attribute to each of the ten function definitions.
- If you apply a namespace, you cannot also specify an access control specifier because namespaces and access control specifiers are mutually exclusive. In other words, you cannot declare a function or property as `public`, `private`, `protected`, or `internal` in addition to applying your namespace.

Referencing namespaces

There is no need to explicitly reference a namespace when you use a method or property declared with any of the access control namespaces, such as `public`, `private`, `protected`, and `internal`. This is because access to these special namespaces is controlled by context. For example, definitions placed into the `private` namespace are automatically available to code within the same class. For namespaces that you define, however, such context sensitivity does not exist. In order to use a method or property that you have placed into a custom namespace, you must reference the namespace.

You can reference namespaces with the `use namespace` directive or you can qualify the name with the namespace using the name qualifier (`::`) punctuator. Referencing a namespace with the `use namespace` directive "opens" the namespace, so that it can apply to any identifiers that are not qualified. For example, if you have defined the `example1` namespace, you can access names in that namespace by using `use namespace example1`:

```
use namespace example1;
myFunction();
```

You can open more than one namespace at a time. Once you open a namespace with `use namespace`, it remains open throughout the block of code in which it was opened. There is no way to explicitly close a namespace.

Having more than one open namespace, however, increases the likelihood of name conflicts. If you prefer not to open a namespace, you can avoid the `use namespace` directive by qualifying the method or property name with the namespace and the name qualifier punctuator. For example, the following code shows how you can qualify the name `myFunction()` with the `example1` namespace:

```
example1::myFunction();
```

Using namespaces

You can find a real-world example of a namespace that is used to prevent name conflicts in the flash.utils.Proxy class that is part of the Flash Player API. The Proxy class, which is the replacement for the `Object.__resolve` property from ActionScript 2.0, allows you to intercept references to undefined properties or methods before an error occurs. All of the methods of the Proxy class reside in the `flash_proxy` namespace in order to prevent name conflicts.

To better understand how the `flash_proxy` namespace is used, you need to understand how to use the Proxy class. The functionality of the Proxy class is available only to classes that inherit from it. In other words, if you want to use the methods of the Proxy class on an object, the object's class definition must extend the Proxy class. For example, if you want to intercept attempts to call an undefined method, you would extend the Proxy class and then override the `callProperty()` method of the Proxy class.

You may recall that implementing namespaces is usually a three-step process of defining, applying, and then referencing a namespace. Because you never explicitly call any of the Proxy class methods, however, the `flash_proxy` namespace is only defined and applied, but never referenced. The Flash Player API defines the `flash_proxy` namespace and applies it in the Proxy class. Your code only needs to apply the `flash_proxy` namespace to classes that extend the Proxy class.

The `flash_proxy` namespace is defined in the flash.utils package in a manner similar to the following:

```
package flash.utils
{
  public namespace flash_proxy;
}
```

The namespace is applied to the methods of the Proxy class as shown in the following excerpt from the Proxy class:

```
public class Proxy
{
  flash_proxy function callProperty(name:*, ... rest):*
  flash_proxy function deleteProperty(name:*):Boolean
  ...
}
```

As the following code shows, you must first import both the Proxy class and the `flash_proxy` namespace. You must then declare your class such that it extends the Proxy class (you must also add the `dynamic` attribute if you are compiling in strict mode). When you override the `callProperty()` method, you must use the `flash_proxy` namespace.

```
package
{
  import flash.utils.Proxy;
  import flash.utils.flash_proxy;

  dynamic class MyProxy extends Proxy
  {
    flash_proxy override function callProperty(name:*, ...rest):*
    {
      trace("method call intercepted: " + name);
    }
  }
}
```

If you create an instance of the MyProxy class and call an undefined method, such as the `testing()` method called in the following example, your Proxy object intercepts the method call and executes the statements inside the overridden `callProperty()` method (in this case, a simple `trace()` statement).

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // method call intercepted: testing
```

There are two advantages to having the methods of the Proxy class inside the `flash_proxy` namespace. First, having a separate namespace reduces clutter in the public interface of any class that extends the Proxy class. (There are about a dozen methods in the Proxy class that you can override, all of which are not designed to be called directly. Placing all of them in the public namespace could be confusing.) Second, use of the `flash_proxy` namespace avoids name conflicts in case your Proxy subclass contains instance methods with names that match any of the Proxy class methods. For example, you may want to name one of your own methods `callProperty()`. The following code is acceptable because your version of the `callProperty()` method is in a different namespace:

```
dynamic class MyProxy extends Proxy
{
  public function callProperty() {}
  flash_proxy override function callProperty(name:*, ...rest):*
  {
    trace("method call intercepted: " + name);
  }
}
```

Namespaces can also be helpful when you want to provide access to methods or properties in a way that cannot be accomplished with the four access control specifiers (`public`, `private`, `internal`, and `protected`). For example, you may have a few utility methods that are spread out across several packages. You want these methods available to all of your packages, but you don't want the methods to be public. To accomplish this, you can create a new namespace and use it as your own special access control specifier.

The following example uses a user-defined namespace to group together two functions that reside in different packages. By grouping them into the same namespace, you can make both functions visible to a class or package through a single `use namespace` statement.

This example uses four files to demonstrate the technique. All of the files must be within your classpath. The first file, myInternal.as, is used to define the `myInternal` namespace. Because the file is in a package named example, you must place the file into a folder named example. The namespace is marked as `public` so that it can be imported into other packages.

```
// myInternal.as in folder example
package example
{
  public namespace myInternal = "http://www.adobe.com/2006/actionscript/
  examples";
}
```

The second and third files, Utility.as and Helper.as, define the classes that contain methods that should be available to other packages. The Utility class is in the example.alpha package, which means that the file should be placed inside a folder named alpha that is a subfolder of the example folder. The Helper class is in the example.beta package, which means that the file should be placed inside a folder named beta that is also a subfolder of the example folder. Both of these packages, example.alpha and example.beta, must import the namespace before using it.

```
// Utility.as in the example/alpha folder
package example.alpha
{
  import example.myInternal;
  use namespace myInternal;

  public class Utility
  {
    private static var _taskCounter:int = 0;

    public static function someTask()
    {
      _taskCounter++;
    }

    myInternal static function get taskCounter():int
    {
      return _taskCounter;
    }
  }
}

// Helper.as in the example/beta folder
package example.beta
{
  import example.myInternal;
  use namespace myInternal;

  public class Helper
  {
    private static var _timeStamp:Date;

    public static function someTask()
    {
      _timeStamp = new Date();
    }

    myInternal static function get lastCalled():Date
    {
      return _timeStamp;
    }
  }
}
```

The fourth file, NamespaceUseCase.as, is the main application class, and should be a sibling to the example folder. The NamespaceUseCase class will also import the `myInternal` namespace and use it to call the two static methods that reside in the other packages. The example uses static methods only to simplify the code. Both static and instance methods can be placed in the `myInternal` namespace.

```
// NamespaceUseCase.as
package
{
  import flash.display.MovieClip;
  import example.myInternal;       // import namespace
  import example.alpha.Utility;    // import Utility class
  import example.beta.Helper;      // import Helper class

  use namespace myInternal;

  public class NamespaceUseCase extends MovieClip
  {
    public function NamespaceUseCase()
    {
      Utility.someTask();
      Utility.someTask();
      trace(Utility.taskCounter); // 2

      Helper.someTask();
      trace(Helper.lastCalled);   // [time someTask() last called]
    }
  }
}
```

Variables

Variables allow you to store values that you use in your program. To declare a variable, you must use the `var` statement with the variable name. In ActionScript 2.0, use of the `var` statement is only required if you use type annotations. In ActionScript 3.0, use of the `var` statement is always required. For example, the following line of ActionScript declares a variable named `i`:

```
var i;
```

If you omit the `var` statement when declaring a variable, you will get a compiler error in strict mode and run-time error in standard mode. For example, the following line of code will result in an error if the variable `i` has not been previously defined:

```
i; // error if i was not previously defined
```

To associate a variable with a data type, you must do so when you declare the variable. Declaring a variable without designating the variable's type is legal, but will generate a compiler warning in strict mode. You designate a variable's type by appending the variable name with a colon (:), followed by the variable's type. For example, the following code declares a variable `i` that is of type int:

```
var i:int;
```

You can assign a value to a variable using the assignment operator (=). For example, the following code declares a variable `i` and assigns the value 20 to it:

```
var i:int;
i = 20;
```

You may find it more convenient to assign a value to a variable at the same time that you declare the variable, as in the following example:

```
var i:int = 20;
```

The technique of assigning a value to a variable at the time it is declared is commonly used not only when assigning primitive values such as integers and strings, but also when creating an array or instantiating an instance of a class. The following example shows an array that is declared and assigned a value using one line of code.

```
var numArray:Array = ["zero", "one", "two"];
```

You can create an instance of a class by using the `new` operator. The following example creates an instance of a named `CustomClass`, and assigns a reference to the newly created class instance to the variable named `customItem`:

```
var customItem:CustomClass = new CustomClass();
```

If you have more than one variable to declare, you can declare them all on one line of code by using the comma operator (`,`) to separate the variables. For example, the following code declares three variables on one line of code:

```
var a:int, b:int, c:int;
```

You can also assign values to each of the variables on the same line of code. For example, the following code declares three variables (`a`, `b`, and `c`) and assigns each a value:

```
var a:int = 10, b:int = 20, c:int = 30;
```

Although you can use the comma operator to group variable declarations into one statement, doing so may reduce the readability of your code.

Understanding variable scope

The *scope* of a variable is the area of your code where the variable can be accessed by a lexical reference. A *global* variable is one that is defined in all areas of your code, whereas a *local* variable is one that is defined in only one part of your code. In ActionScript 3.0, variables are always scoped to the function or class in which they are declared. A global variable is a variable that you define outside of any function or class definition. For example, the following code creates a global variable `strGlobal` by declaring it outside of any function. The example shows that a global variable is available both inside and outside the function definition.

```
var strGlobal:String = "Global";
function scopeTest ()
{
  trace(strGlobal); // Global
}
scopeTest();
trace(strGlobal); // Global
```

You declare a local variable by declaring the variable inside a function definition. The smallest area of code for which you can define a local variable is a function definition. A local variable declared within a function will exist only in that function. For example, if you declare a variable named `str2` within a function named `localScope()`, that variable will not be available outside the function.

```
function localScope()
{
  var strLocal:String = "local";
}
localScope();
trace(strLocal); // error because strLocal is not defined globally
```

If the variable name you use for your local variable is already declared as a global variable, the local definition hides (or shadows) the global definition while the local variable is in scope. The global variable will still exist outside of the function. For example, the following code creates a global string variable named `str1`, and then creates a local variable of the same name inside the `scopeTest()` function. The `trace` statement inside the function outputs the local value of the variable, but the `trace` statement outside the function outputs the global value of the variable.

```
var str1:String = "Global";
function scopeTest ()
{
  var str1:String = "Local";
  trace(str1); // Local
}
scopeTest();
trace(str1); // Global
```

ActionScript variables, unlike variables in C++ and Java, do not have block-level scope. A block of code is any group of statements between an opening curly brace (`{`) and a closing curly brace (`}`). In some programming languages, such as C++ and Java, variables declared inside a block of code are not available outside that block of code. This restriction of scope is called block-level scope, and does not exist in ActionScript. If you declare a variable inside a block of code, that variable will be available not only in that block of code, but also in any other parts of the function to which the code block belongs. For example, the following function contains variables that are defined in various block scopes. All the variables are available throughout the function.

```
function blockTest (testArray:Array)
{
  var numElements:int = testArray.length;
  if (numElements > 0)
  {
    var elemStr:String = "Element #";
    for (var i:int = 0; i < numElements; i++)
    {
      var valueStr:String = i + ": " + testArray[i];
      trace(elemStr + valueStr);
    }
    trace(elemStr, valueStr, i);   // all still defined
  }
  trace(elemStr, valueStr, i); // all defined if numElements > 0
}

blockTest(["Earth", "Moon", "Sun"]);
```

An interesting implication of the lack of block-level scope is that you can read or write to a variable before it is declared, as long as it is declared before the function ends. This is because of a technique called *hoisting*, which means that the compiler moves all variable declarations to the top of the function. For example, the following code compiles even though the initial `trace()` function for the `num` variable happens before the `num` variable is declared:

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

The compiler will not, however, hoist any assignment statements. This explains why the initial `trace()` of `num` results in `NaN` (not a number), which is the default value for variables of the Number data type. This means that you can assign values to variables even before they are declared, as shown in the following example:

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

Default values

A *default value* is the value that a variable contains before you set its value. You *initialize* a variable when you set its value for the first time. If you declare a variable, but do not set its value, that variable is *uninitialized*. The value of an uninitialized variable depends on its data type. The following table describes the default values of variables, organized by data type:

Data type	Default value
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
String	`null`
uint	0
Not declared (equivalent to type annotation `*`)	`undefined`
All other classes, including user-defined classes.	`null`

For variables of type Number, the default value is `NaN` (not a number), which is a special value defined by the IEEE-754 standard to mean a value that does not represent a number.

If you declare a variable, but do not declare its data type, the default data type `*` will apply, which actually means that the variable is untyped. If you also do not initialize an untyped variable with a value, its default value is `undefined`.

For data types other than Boolean, Number, int, and uint, the default value of any uninitialized variable is `null`. This applies to all the classes defined by the Flash Player API, as well as any custom classes that you create.

The value `null` is not a valid value for variables of type Boolean, Number, int, or uint. If you attempt to assign a value of `null` to a such a variable, the value is converted to the default value for that data type. For variables of type Object, you can assign a value of `null`. If you attempt to assign the value `undefined` to a variable of type Object, the value is converted to `null`.

For variables of type Number, there is a special top-level function named `isNaN()` that returns the Boolean value `true` if the variable is not a number, and `false` otherwise.

Data types

A *data type* defines a set of values. For example, the Boolean data type is the set of exactly two values: `true` and `false`. In addition to the Boolean data type, ActionScript 3.0 defines several more commonly used data types, such as String, Number, and Array. You can define your own data types by using classes or interfaces to define a custom set of values. All values in ActionScript 3.0, whether they are primitive or complex, are objects.

A *primitive value* is a value that belongs to one of the following data types: Boolean, int, Number, String, and uint. Working with primitive values is usually faster than working with complex values because ActionScript stores primitive values in a special way that makes memory and speed optimizations possible.

NOTE

For readers interested in the technical details, ActionScript stores primitive values internally as immutable objects. The fact that they are stored as immutable objects means that passing by reference is effectively the same as passing by value. This cuts down on memory usage and increases execution speed because references are usually significantly smaller than the values themselves.

A *complex value* is a value that is not a primitive value. Data types that define sets of complex values include Array, Date, Error, Function, RegExp, XML, and XMLList.

Many programming languages distinguish between primitive values and their wrapper objects. Java, for example, has an int primitive and the java.lang.Integer class that wraps it. Java primitives are not objects, but their wrappers are, which makes primitives useful for some operations and wrapper objects better suited for other operations. In ActionScript 3.0, primitive values and their wrapper objects are, for practical purposes, indistinguishable. All values, even primitive values, are objects. Flash Player treats these primitive types as special cases that behave like objects but that don't require the normal overhead associated with creating objects. This means that the following two lines of code are equivalent:

```
var someInt:int = 3;
var someInt:int = new int(3);
```

All the primitive and complex data types listed above are defined by the ActionScript 3.0 core classes. The core classes allow you to create objects using literal values instead of using the `new` operator. For example, you can create an array using a literal value or the Array class constructor, as follows:

```
var someArray:Array = [1, 2, 3]; // literal value
var someArray:Array = new Array(1,2,3); // Array constructor
```

Type checking

Type checking can occur at either compile time or run time. Statically typed languages, such as C++ and Java, do type checking at compile time. Dynamically typed languages, such as Smalltalk and Python, handle type checking at run time. As a dynamically typed language, ActionScript 3.0 has run-time type checking, but also supports compile-time type checking with a special compiler mode called *strict mode*. In strict mode, type checking occurs at both compile time and run time, but in standard mode, type checking occurs only at run time.

Dynamically typed languages offer tremendous flexibility when you structure your code, but at the cost of allowing type errors to manifest at run time. Statically typed languages report type errors at compile time, but at the cost of requiring type information to be known at compile time.

Compile-time type checking

Compile-time type checking is often favored in larger projects because as the size of a project grows, data type flexibility usually becomes less important than catching type errors as early as possible. This is why, by default, the ActionScript compiler in Adobe Flash CS3 and Adobe Flex Builder 2 is set to run in strict mode. You can disable strict mode in Flex Builder 2 through the ActionScript compiler settings in the Project Properties dialog box.

In order to provide compile-time type checking, the compiler needs to know the data type information for the variables or expressions in your code. To explicitly declare a data type for a variable, add the colon operator (`:`) followed by the data type as a suffix to the variable name. To associate a data type with a parameter, you use the colon operator followed by the data type. For example, the following code adds data type information to the `xParam` parameter, and declares a variable `myParam` with an explicit data type:

```
function runtimeTest(xParam:String)
{
  trace(xParam);
}
var myParam:String = “hello”;
runtimeTest(myParam);
```

In strict mode, the ActionScript compiler reports type mismatches as compiler errors. For example, the following code declares a function parameter `xParam`, of type Object, but later attempts to assign values of type String and Number to that parameter. This produces a compiler error in strict mode.

```
function dynamicTest(xParam:Object)
{
  if (xParam is String)
  {
    var myStr:String = xParam; // compiler error in strict mode
    trace("String: " + myStr);
  }
  else if (xParam is Number)
  {
    var myNum:Number = xParam; // compiler error in strict mode
    trace("Number: " + myNum);
  }
}
```

Even in strict mode, however, you can selectively opt of out compile-time type checking by leaving the right side of an assignment statement untyped. You can mark a variable or expression as untyped by either omitting a type annotation, or using the special asterisk (`*`) type annotation. For example, if the `xParam` parameter in the previous example is modified so that it no longer has a type annotation, the code will compile in strict mode:

```
function dynamicTest(xParam)
{
  if (xParam is String)
  {
    var myStr:String = xParam;
    trace("String: " + myStr);
  }
  else if (xParam is Number)
  {
    var myNum:Number = xParam;
    trace("Number: " + myNum);
  }
}
dynamicTest(100)
dynamicTest("one hundred");
```

Run-time type checking

Run-time type checking occurs in ActionScript 3.0 whether you compile in strict mode or standard mode. Consider a situation in which the value 3 is passed as an argument to a function that expects an array. In strict mode, the compiler will generate an error because the value 3 is not compatible with the data type Array. If you disable strict mode, and run in standard mode, the compiler does not complain about the type mismatch, but run-time type checking by Flash Player results in a run-time error.

The following example shows a function named `typeTest()` that expects an Array argument but is passed a value of 3. This causes a run-time error in standard mode because the value 3 is not a member of the parameter's declared data type (Array).

```
function typeTest(xParam:Array)
{
  trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0 standard mode
```

There may also be situations where you get a run-time type error even when you are operating in strict mode. This is possible if you use strict mode, but opt out of compile-time type checking by using an untyped variable. When you use an untyped variable, you are not eliminating type checking, but rather deferring it until run time. For example, if the `myNum` variable in the previous example does not have a declared data type, the compiler cannot detect the type mismatch, but Flash Player will generate a run-time error because it compares the run-time value of `myNum`, which is set to 3 as a result of the assignment statement, with the type of `xParam`, which is set to the Array data type.

```
function typeTest(xParam:Array)
{
  trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0
```

Run-time type checking also allows more flexible use of inheritance than does compile-time checking. By deferring type checking to run time, standard mode allows you to reference properties of a subclass even if you *upcast*. An upcast occurs when you use a base class to declare the type of a class instance, but a subclass to instantiate it. For example, you can create a class named ClassBase that can be extended (classes with the `final` attribute cannot be extended):

```
class ClassBase
{
}
```

You can subsequently create a subclass of ClassBase named ClassExtender, which has one property named `someString`, as follows:

```
class ClassExtender extends ClassBase
{
  var someString:String;
}
```

Using both classes, you can create a class instance that is declared using the ClassBase data type, but instantiated using the ClassExtender constructor. Upcasts are considered safe operations because the base class does not contain any properties or methods that are not in the subclass.

```
var myClass:ClassBase = new ClassExtender();
```

A subclass, however, does contain properties or methods that its base class does not. For example, the ClassExtender class contains the `someString` property, which does not exist in the ClassBase class. In ActionScript 3.0 standard mode, you can reference this property using the `myClass` instance without generating a compile-time error, as shown in the following example:

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// no error in ActionScript 3.0 standard mode
```

The is operator

The `is` operator, which is new for ActionScript 3.0, allows you to test whether a variable or expression is a member of a given data type. In previous versions of ActionScript, the `instanceof` operator provided this functionality, but in ActionScript 3.0 the `instanceof` operator should not be used to test for data type membership. The `is` operator should be used instead of the `instanceof` operator for manual type checking because the expression `x instanceof y` merely checks the prototype chain of `x` for the existence of `y` (and in ActionScript 3.0, the prototype chain does not provide a complete picture of the inheritance hierarchy).

The `is` operator examines the proper inheritance hierarchy and can be used to check not only whether an object is an instance of a particular class, but also whether an object is an instance of a class that implements a particular interface. The following example creates an instance of the Sprite class named `mySprite` and uses the `is` operator to test whether `mySprite` is an instance of the Sprite and DisplayObject classes, and whether it implements the IEventDispatcher interface:

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite);                   // true
trace(mySprite is DisplayObject);            // true
trace(mySprite is IEventDispatcher);         // true
```

The `is` operator checks the inheritance hierarchy and properly reports that `mySprite` is compatible with the Sprite and DisplayObject classes (the Sprite class is a subclass of the DisplayObject class). The `is` operator also checks whether `mySprite` inherits from any classes that implement the IEventDispatcher interface. Because the Sprite class inherits from the EventDispatcher class, which implements the IEventDispatcher interface, the `is` operator correctly reports that `mySprite` implements the same interface.

The following example shows the same tests from the previous example, but with `instanceof` instead of the `is` operator. The `instanceof` operator correctly identifies that `mySprite` is an instance of Sprite or DisplayObject, but it returns `false` when used to test whether `mySprite` implements the IEventDispatcher interface.

```
trace(mySprite instanceof Sprite);           // true
trace(mySprite instanceof DisplayObject);    // true
trace(mySprite instanceof IEventDispatcher); // false
```

The as operator

The `as` operator, which is new in ActionScript 3.0, also allows you to check whether an expression is a member of a given data type. Unlike the `is` operator, however, the `as` operator does not return a Boolean value. Rather, the `as` operator returns the value of the expression instead of `true`, and `null` instead of `false`. The following example shows the results of using the `as` operator instead of the `is` operator in the simple case of checking whether a Sprite instance is a member of the DisplayObject, IEventDispatcher, and Number data types.

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [object Sprite]
trace(mySprite as DisplayObject); // [object Sprite]
trace(mySprite as IEventDispatcher); // [object Sprite]
trace(mySprite as Number);   // null
```

When you use the `as` operator, the operand on the right must be a data type. An attempt to use an expression other than a data type as the operand on the right will result in an error.

Dynamic classes

A *dynamic* class defines an object that can be altered at run time by adding or changing properties and methods. A class that is not dynamic, such as the String class, is a *sealed* class. You cannot add properties or methods to a sealed class at run time.

You create dynamic classes by using the `dynamic` attribute when you declare a class. For example, the following code creates a dynamic class named `Protean`:

```
dynamic class Protean
{
  private var privateGreeting:String = "hi";
  public var publicGreeting:String = "hello";
  function Protean()
  {
    trace("Protean instance created");
  }
}
```

If you subsequently instantiate an instance of the `Protean` class, you can add properties or methods to it outside the class definition. For example, the following code creates an instance of the `Protean` class and adds a property named `aString` and a property named `aNumber` to the instance:

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // testing 3
```

Properties that you add to an instance of a dynamic class are run-time entities, so any type checking is done at run time. You cannot add a type annotation to a property that you add in this manner.

You can also add a method to the `myProtean` instance by defining a function and attaching the function to a property of the `myProtean` instance. The following code moves the trace statement into a method named `traceProtean()`:

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
  trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // testing 3
```

Methods created in this way, however, do not have access to any private properties or methods of the Protean class. Moreover, even references to public properties or methods of the `Protean` class must be qualified with either the `this` keyword or the class name. The following example shows the `traceProtean()` method attempting to access the private and public variables of the `Protean` class.

```
myProtean.traceProtean = function ()
{
  trace(myProtean.privateGreeting); // undefined
  trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

Data type descriptions

The primitive data types include Boolean, int, Null, Number, String, uint, and void. The ActionScript core classes also define the following complex data types: Object, Array, Date, Error, Function, RegExp, XML, and XMLList.

Boolean data type

The Boolean data type comprises two values: `true` and `false`. No other values are valid for variables of Boolean type. The default value of a Boolean variable that has been declared but not initialized is `false`.

int data type

The int data type is stored internally as a 32-bit integer and comprises the set of integers from -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$), inclusive. Previous versions of ActionScript offered only the Number data type, which was used for both integers and floating point numbers. In ActionScript 3.0, you now have access to low-level machine types for 32-bit signed and unsigned integers. If your variable will not use floating point numbers, using the int data type instead of the Number data type should be faster and more efficient.

For integer values outside the range of the minimum and maximum int values, use the Number data type, which can handle values between positive and negative 9,007,199,254,740,992 (53-bit integer values). The default value for variables that are of the data type int is 0.

Null data type

The Null data type contains only one value, `null`. This is the default value for the String data type and all classes that define complex data types, including the Object class. None of the other primitive data types, such as Boolean, Number, int and uint, contain the value `null`. Flash Player will convert the value `null` to the appropriate default value if you attempt to assign `null` to variables of type Boolean, Number, int, or uint. You cannot use this data type as a type annotation.

Number data type

In ActionScript 3.0, the Number data type can represent integers, unsigned integers, and floating point numbers. However, to maximize performance, you should use the Number data type only for integer values larger than the 32-bit `int` and `uint` types can store or for floating point numbers. To store a floating point number, include a decimal point in the number. If you omit a decimal point, the number will be stored as an integer.

The Number data type uses the 64-bit double-precision format as specified by the IEEE Standard for Binary Floating-Point Arithmetic (IEEE-754). This standard dictates how floating point numbers are stored using the 64 available bits. One bit is used to designate whether the number is positive or negative. Eleven bits are used for the exponent, which is stored as base 2. The remaining 52 bits are used to store the *significand* (also called the *mantissa*), which is the number that is raised to the power indicated by the exponent.

By using some of its bits to store an exponent, the Number data type can store floating point numbers significantly larger than if it used all of its bits for the significand. For example, if the Number data type used all 64 bits to store the significand, it could store a number as large as 2^{64}. By using 11 bits to store an exponent, the Number data type can raise its significand to a power of 2^{1023}.

The maximum and minimum values that the Number type can represent are stored in static properties of the Number class called `Number.MAX_VALUE` and `Number.MIN_VALUE`.

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

Although this range of numbers is enormous, the cost of this range is precision. The Number data type uses 52 bits to store the significand, with the result that numbers that require more than 52 bits to represent precisely, such as the fraction 1/3, are only approximations. If your application requires absolute precision with decimal numbers, you need to use software that implements decimal floating point arithmetic as opposed to binary floating point arithmetic.

When you store integer values with the Number data type, only the 52 bits of the significand are used. The Number data type uses these 52 bits and a special hidden bit to represent integers from -9,007,199,254,740,992 (-2^{53}) to 9,007,199,254,740,992 (2^{53}).

Flash Player uses the `NaN` value not only as the default value for variables of type `Number`, but also as the result of any operation that should return a number but does not. For example, if you attempt to calculate the square root of a negative number, the result will be `NaN`. Other special Number values include *positive infinity* and *negative infinity*.

NOTE

The result of division by `0` is only `NaN` if the divisor is also `0`. Division by `0` produces `infinity` when the dividend is positive or `-infinity` when the dividend is negative.

String data type

The String data type represents a sequence of 16-bit characters. Strings are stored internally as Unicode characters, using the UTF-16 format. Strings are immutable values, just as they are in the Java programming language. An operation on a String value returns a new instance of the string. The default value for a variable declared with the String data type is `null`. The value `null` is not the same as the empty string (`""`), even though they both represent the absence of any characters.

uint data type

The uint data type is stored internally as a 32-bit unsigned integer and comprises the set of integers from 0 to 4,294,967,295 ($2^{32}-1$), inclusive. Use the uint data type for special circumstances that call for non-negative integers. For example, you must use the uint data type to represent pixel color values because the int data type has an internal sign bit that is not appropriate for handling color values. For integer values larger than the maximum uint value, use the Number data type, which can handle 53-bit integer values. The default value for variables that are of data type uint is 0.

void data type

The void data type contains only one value, `undefined`. In previous versions of ActionScript, `undefined` was the default value for instances of the Object class. In ActionScript 3.0, the default value for Object instances is `null`. If you attempt to assign the value `undefined` to an instance of the Object class, Flash Player will convert the value to `null`. You can only assign a value of `undefined` to variables that are untyped. Untyped variables are variables that either lack any type annotation, or use the asterisk (`*`) symbol for type annotation. You can use `void` only as a return type annotation.

Object data type

The Object data type is defined by the Object class. The Object class serves as the base class for all class definitions in ActionScript. The ActionScript 3.0 version of the Object data type differs from that of previous versions in three ways. First, the Object data type is no longer the default data type assigned to variables with no type annotation. Second, the Object data type no longer includes the value `undefined`, which used to be the default value of Object instances. Third, in ActionScript 3.0, the default value for instances of the Object class is `null`.

In previous versions of ActionScript, a variable with no type annotation was automatically assigned the Object data type. This is no longer true in ActionScript 3.0, which now includes the idea of a truly untyped variable. Variables with no type annotation are now considered untyped. If you prefer to make it clear to readers of your code that your intention is to leave a variable untyped, you can use the new asterisk (`*`) symbol for the type annotation, which is equivalent to omitting a type annotation. The following example shows two equivalent statements, both of which declare an untyped variable `x`:

```
var x
var x:*
```

Only untyped variables can hold the value `undefined`. If you attempt to assign the value `undefined` to a variable that has a data type, Flash Player will convert the value `undefined` to the default value of that data type. For instances of the Object data type, the default value is `null`, which means that Flash Player will convert the value `undefined` to `null` if you attempt to assign `undefined` to an Object instance.

Type conversions

A type conversion is said to occur when a value is transformed into a value of a different data type. Type conversions can be either *implicit* or *explicit*. Implicit conversion, which is also called *coercion*, is sometimes performed by Flash Player at run time. For example, if the value 2 is assigned to a variable of the Boolean data type, Flash Player converts the value 2 to the Boolean value `true` before assigning the value to the variable. Explicit conversion, which is also called *casting*, occurs when your code instructs the compiler to treat a variable of one data type as if it belongs to a different data type. When primitive values are involved, casting actually converts values from one data type to another. To cast an object to a different type, you wrap the object name in parentheses and precede it with the name of the new type. For example, the following code takes a Boolean value and casts it to an integer:

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

Implicit conversions

Implicit conversions happen at run time in a number of contexts:

- In assignment statements
- When values are passed as function arguments
- When values are returned from functions
- In expressions using certain operators, such as the addition (+) operator

For user-defined types, implicit conversions succeed when the value to be converted is an instance of the destination class or a class that derives from the destination class. If an implicit conversion is unsuccessful, an error occurs. For example, the following code contains a successful implicit conversion and an unsuccessful implicit conversion:

```
class A {}
class B extends A {}

var objA:A = new A();
var objB:B = new B();
var arr:Array = new Array();

objA = objB; // Conversion succeeds.
objB = arr; // Conversion fails.
```

For primitive types, implicit conversions are handled by calling the same internal conversion algorithms that are called by the explicit conversion functions. The following sections discuss these primitive type conversions in detail.

Explicit conversions

It's helpful to use explicit conversions, or casting, when you compile in strict mode because there may be times when you do not want a type mismatch to generate a compile-time error. This may be the case when you know that coercion will convert your values correctly at run time. For example, when working with data received from a form, you may want to rely on coercion to convert certain string values to numeric values. The following code generates a compile-time error even though the code would run correctly in standard mode.

```
var quantityField:String = "3";
var quantity:int = quantityField; // compile time error in strict mode
```

If you want to continue using strict mode, but would like the string converted to an integer, you can use explicit conversion, as follows:

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // Explicit conversion succeeds.
```

Casting to int, uint, and Number

You can cast any data type into one of the three number types: int, uint, and Number. If Flash Player is unable to convert the number for some reason, the default value of 0 is assigned for the int and uint data types, and the default value of `NaN` is assigned for the Number data type. If you convert a Boolean value to a number, `true` becomes the value 1 and `false` becomes the value 0.

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

String values that contain only digits can be successfully converted into one of the number types. The number types can also convert strings that look like negative numbers or strings that represent a hexadecimal value (for example, `0x1A`). The conversion process ignores leading and trailing white space characters in the string value. You can also cast strings that look like floating point numbers using `Number()`. The inclusion of a decimal point causes `uint()` and `int()` to return an integer with the characters following the decimal that has been truncated. For example, the following string values can be cast into numbers.

```
trace(uint("5"));      // 5
trace(uint("-5"));     // 4294967291. It wraps around from MAX_VALUE
trace(uint(" 27 "));   // 27
trace(uint("3.7"));    // 3
trace(int("3.7"));     // 3
trace(int("0x1A"));    // 26
trace(Number("3.7")); // 3.7
```

String values that contain non-numeric characters return 0 when cast with `int()` or `uint()` and `NaN` when case with `Number()`. The conversion process ignores leading and trailing white space, but returns 0 or `NaN` if a string has white space separating two numbers.

```
trace(uint("5a"));     // 0
trace(uint("ten"));    // 0
trace(uint("17 63")); // 0
```

In ActionScript 3.0, the `Number()` function no longer supports octal, or base 8, numbers. If you supply a string with a leading zero to the ActionScript 2.0 `Number()` function, the number is interpreted as an octal number, and converted to its decimal equivalent. This is not true with the `Number()` function in ActionScript 3.0, which instead ignores the leading zero. For example, the following code generates different output when compiled using different versions of ActionScript:

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

Casting is not necessary when a value of one numeric type is assigned to a variable of a different numeric type. Even in strict mode, the numeric types are implicitly converted to the other numeric types. This means that in some cases, unexpected values may result when the range of a type is exceeded. The following examples all compile in strict mode, though some will generate unexpected values:

```
var sampleUINT:uint = -3; // Assign value of type int and Number.
trace(sampleUINT); // 4294967293

var sampleNum:Number = sampleUINT; // Assign value of type int and uint.
trace(sampleNum) // 4294967293

var sampleINT:int = uint.MAX_VALUE + 1; // Assign value of type Number.
trace(sampleINT); // 0

sampleINT = int.MAX_VALUE + 1; // Assign value of type uint and Number.
trace(sampleINT); // -2147483648
```

The following table summarizes the results of casting to the Number, int, or uint data type from other data types.

Data type or value	Result of conversion to Number, int or uint
Boolean	If the value is `true`, 1; otherwise, 0.
Date	The internal representation of the Date object, which is the number of milliseconds since midnight January 1, 1970, universal time.
`null`	0
Object	If the instance is `null` and converted to Number, `NaN`; otherwise, 0.
String	A number if Flash Player can convert the string to a number; otherwise, `NaN` if converted to Number or 0 if converted to int or uint.
`undefined`	If converted to Number, `NaN`; if converted to int or uint, 0.

Casting to Boolean

Casting to Boolean from any of the numeric data types (uint, int, and Number) results in `false` if the numeric value is 0, and `true` otherwise. For the Number data type, the value `NaN` also results in `false`. The following example shows the results of casting the numbers -1, 0, and 1:

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
  trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

The output from the example shows that of the three numbers, only 0 returns a value of `false`:

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

Casting to Boolean from a String value returns `false` if the string is either `null` or an empty string (""). Otherwise, it returns `true`.

```
var str1:String;       // Uninitialized string is null.
trace(Boolean(str1));  // false

var str2:String = "";  // empty string
trace(Boolean(str2));  // false

var str3:String = " "; // white space only
trace(Boolean(str3));  // true
```

Casting to Boolean from an instance of the Object class returns `false` if the instance is `null`, and `true` otherwise, as the following example shows:

```
var myObj:Object;      // Uninitialized object is null.
trace(Boolean(myObj)); // false

myObj = new Object();  // instantiate
trace(Boolean(myObj)); // true
```

Boolean variables get special treatment in strict mode in that you can assign values of any data type to a Boolean variable without casting. Implicit coercion from all data types to the Boolean data type occurs even in strict mode. In other words, unlike almost all other data types, casting to Boolean is not necessary to avoid strict mode errors. The following examples all compile in strict mode and behave as expected at run time:

```
var myObj:Object = new Object();  // instantiate
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

The following table summarizes the results of casting to the Boolean data type from other data types:

Data type or value	Result of conversion to Boolean
String	`false` if the value is `null` or the empty string (`""`); `true` otherwise.
`null`	`false`
Number, int or uint	`false` if the value is NaN or 0; `true` otherwise.
Object	`false` if the instance is null; `true` otherwise.

Casting to String

Casting to the String data type from any of the numeric data types returns a string representation of the number. Casting to the String data type from a Boolean value returns the string "`true`" if the value is `true`, and returns the string "`false`" if the value is `false`.

Casting to the String data type from an instance of the Object class returns the string "`null`" if the instance is `null`. Otherwise, casting to the String type from the Object class returns the string "`[object Object]`".

Casting to String from an instance of the Array class returns a string comprising a comma-delimited list of all the array elements. For example, the following cast to the String data type returns one string containing all three elements of the array:

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

Casting to String from an instance of the Date class returns a string representation of the date that the instance contains. For example, the following example returns a string representation of the Date class instance (the output shows result for Pacific Daylight Time):

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // Fri Jul 1 00:00:00 GMT-0700 2005
```

The following table summarizes the results of casting to the String data type from other data types.

Data type or value	Result of conversion to string
Array	A string comprising all array elements.
Boolean	`"true"` or `"false"`
Date	A string representation of the Date object.
`null`	`"null"`
Number, int or uint	A string representation of the number.
Object	If the instance is null, `"null"`; otherwise, `"[object Object]".`

Syntax

The syntax of a language defines a set of rules that must be followed when writing executable code.

Case sensitivity

ActionScript 3.0 is a case-sensitive language. Identifiers that differ only in case are considered different identifiers. For example, the following code creates two different variables:

```
var num1:int;
var Num1:int;
```

Dot syntax

The dot operator (`.`) provides a way to access the properties and methods of an object. Using dot syntax, you can refer to a class property or method using an instance name, followed by the dot operator and name of the property or method. For example, consider the following class definition:

```
class DotExample
{
  public var prop1:String;
```

```
    public function method1():void {}
}
```

Using dot syntax, you can access the `prop1` property and the `method1()` method using the instance name created in the following code:

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = “hello”;
myDotEx.method1();
```

You can use dot syntax when you define packages. You use the dot operator to refer to nested packages. For example, the EventDispatcher class resides in a package named events that is nested within the package named flash. You can refer to the events package using the following expression:

```
flash.events
```

You can also refer to the EventDispatcher class using this expression:

```
flash.events.EventDispatcher
```

Slash syntax

Slash syntax is not supported in ActionScript 3.0. Slash syntax was used in earlier versions of ActionScript to indicate the path of a movie clip or variable.

Literals

A *literal* is a value that appears directly in your code. The following examples are all literals:

```
17
"hello"
-3
9.4
null
undefined
true
false
```

Literals can also be grouped to form compound literals. Array literals are enclosed in bracket characters (`[]`) and use the comma to separate array elements.

An array literal can be used to initialize an array. The following examples show two arrays that are initialized using array literals. You can use the `new` statement and pass the compound literal as a parameter to the Array class constructor, but you can also assign literal values directly when instantiating instances of the following ActionScript core classes: Object, Array, String, Number, int, uint, XML, XMLList and Boolean.

```
// Use new statement.
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// Assign literal directly.
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

Literals can also be used to initialize a generic object. A generic object is an instance of the Object class. Object literals are enclosed in curly braces (`{}`) and use the comma to separate object properties. Each property is declared with the colon character (`:`), which separates the name of the property from the value of the property.

You can create a generic object using the `new` statement, and pass the object literal as a parameter to the Object class constructor, or you can assign the object literal directly to the instance you are declaring. The following example creates a new generic object and initializes the object with three properties (`propA`, `propB`, and `propC`), each with values set to 1, 2, and 3, respectively.

```
// Use new statement.
var myObject:Object = new Object({propA:1, propB:2, propC:3});

// Assign literal directly.
var myObject:Object = {propA:1, propB:2, propC:3};
```

For more information, see "Creating strings" on page 209, "Introduction to Regular Expressions" on page 286, and "Initializing XML variables" on page 319.

Semicolons

You can use the semicolon character (`;`) to terminate a statement. Alternatively, if you omit the semicolon character, the compiler will assume that each line of code represents a single statement. Because many programmers are accustomed to using the semicolon to denote the end of a statement, your code may be easier to read if you consistently use semicolons to terminate your statements.

Using a semicolon to terminate a statement allows you to place more than one statement on a single line, but this may make your code more difficult to read.

Parentheses

You can use parentheses (`()`) in three ways in ActionScript 3.0. First, you can use parentheses to change the order of operations in an expression. Operations that are grouped inside parentheses are always executed first. For example, parentheses are used to alter the order of operations in the following code:

```
trace(2 + 3 * 4);    // 14
trace( (2 + 3) * 4); // 20
```

Second, you can use parentheses with the comma operator (`,`) to evaluate a series of expressions and return the result of the final expression, as shown in the following example:

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

Third, you can use parentheses to pass one or more parameters to functions or methods, as shown in the following example, which passes a String value to the `trace()` function:

```
trace("hello"); // hello
```

Comments

ActionScript 3.0 code supports two types of comments: single-line comments and multiline comments. These commenting mechanisms are similar to the commenting mechanisms in C++ and Java. The compiler will ignore text that is marked as a comment.

Single-line comments begin with two forward slash characters (`//`) and continue until the end of the line. For example, the following code contains a single-line comment:

```
var someNumber:Number = 3; // a single line comment
```

Multiline comments begin with a forward slash and asterisk (`/*`) and end with an asterisk and forward slash (`*/`).

```
/* This is multiline comment that can span
more than one line of code. */
```

Keywords and reserved words

Reserved words are words that you cannot use as identifiers in your code because the words are reserved for use by ActionScript. Reserved words include *lexical keywords*, which are removed from the program namespace by the compiler. The compiler will report an error if you use a lexical keyword as an identifier. The following table lists ActionScript 3.0 lexical keywords.

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	function
if	implements	import	in
instanceof	interface	internal	is
native	new	null	package
private	protected	public	return
super	switch	this	throw
to	true	try	typeof
use	var	void	while
with			

There is a small set of keywords, called *syntactic keywords*, that can be used as identifiers, but that have special meaning in certain contexts. The following table lists ActionScript 3.0 syntactic keywords.

each	get	set	namespace
include	dynamic	final	native
override	static		

There are also several identifiers that are sometimes referred to as *future reserved words*. These identifiers are not reserved by ActionScript 3.0, though some of them may be treated as keywords by products that incorporate ActionScript 3.0. You might be able to use many of these identifiers in your code, but Adobe recommends that you do not use them because they may appear as keywords in a subsequent version of the language.

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic
long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

Constants

ActionScript 3.0 supports the `const` statement, which you can use to create constants. Constants are properties with a fixed value that cannot be altered. You can assign a value to a constant only once, and the assignment must occur in close proximity to the declaration of the constant. For example, if a constant is declared as a member of a class, you can assign a value to that constant only as part of the declaration or inside the class constructor.

The following code declares two constants. The first constant, MINIMUM, has a value assigned as part of the declaration statement. The second constant, MAXIMUM, has a value assigned in the constructor.

```
class A
{
  public const MINIMUM:int = 0;
  public const MAXIMUM:int;

  public function A()
  {
    MAXIMUM = 10;
  }
}

var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

An error results if you attempt to assign an initial value to a constant in any other way. For example, if you attempt to set the initial value of MAXIMUM outside the class, a run-time error will occur.

```
class A
{
  public const MINIMUM:int = 0;
  public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // run-time error
```

The Flash Player API defines a wide range of constants for your use. By convention, constants in ActionScript use all capital letters, with words separated by the underscore character (_). For example, the MouseEvent class definition uses this naming convention for its constants, each of which represents an event related to mouse input:

```
package flash.events
{
  public class MouseEvent extends Event
  {
    public static const CLICK:String            = "click";
    public static const DOUBLE_CLICK:String     = "doubleClick";
    public static const MOUSE_DOWN:String       = "mouseDown";
    public static const MOUSE_MOVE:String       = "mouseMove";
    ...
  }
}
```

Operators

Operators are special functions that take one or more operands and return a value. An *operand* is a value—usually a literal, a variable, or an expression—that an operator uses as input. For example, in the following code, the addition (+) and multiplication (*) operators are used with three literal operands (2, 3, and 4) to return a value. This value is then used by the assignment (=) operator to assign the returned value, 14, to the variable sumNumber.

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

Operators can be unary, binary, or ternary. A *unary* operator takes one operand. For example, the increment (++) operator is a unary operator because it takes only one operand. A *binary* operator takes two operands. For example, the division (/) operator takes two operands. A *ternary* operator takes three operands. For example, the conditional (?:) operator takes three operands.

Some operators are *overloaded*, which means that they behave differently depending on the type or quantity of operands passed to them. The addition (+) operator is an example of an overloaded operator that behaves differently depending on the data type of the operands. If both operands are numbers, the addition operator returns the sum of the values. If both operands are strings, the addition operator returns the concatenation of the two operands. The following example code shows how the operator behaves differently depending on the operands.

```
trace(5 + 5);     // 10
trace("5" + "5"); // 55
```

Operators can also behave differently based on the number of operands supplied. The subtraction (-) operator is both a unary and binary operator. When supplied with only one operand, the subtraction operator negates the operand and returns the result. When supplied with two operands, the subtraction operator returns the difference between the operands. The following example shows the subtraction operator used first as a unary operator, and then as a binary operator.

```
trace(-3);  // -3
trace(7-2); // 5
```

Operator precedence and associativity

Operator precedence and associativity determine the order in which operators are processed. Although it may seem natural to those familiar with arithmetic that the compiler processes the multiplication (*) operator before the addition (+) operator, the compiler needs explicit instructions about which operators to process first. Such instructions are collectively referred to as *operator precedence*. ActionScript defines a default operator precedence that you can alter using the parentheses (`()`) operator. For example, the following code alters the default precedence in the previous example to force the compiler to process the addition operator before the multiplication operator:

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

You may encounter situations in which two or more operators of the same precedence appear in the same expression. In these cases, the compiler uses the rules of *associativity* to determine which operator to process first. All of the binary operators, except the assignment operators, are *left-associative*, which means that operators on the left are processed before operators on the right. The assignment operators and the conditional (`?:`) operator are *right-associative*, which means that the operators on the right are processed before operators on the left.

For example, consider the less-than (`<`) and greater-than (`>`) operators, which have the same precedence. If both operators are used in the same expression, the operator on the left is processed first because both operators are left-associative. This means that the following two statements produce the same output:

```
trace(3 > 2 < 1);   // false
trace((3 > 2) < 1); // false
```

The greater-than operator is processed first, which results in a value of `true` because the operand 3 is greater than the operand 2. The value `true` is then passed to the less-than operator, along with the operand 1. The following code represents this intermediate state:

```
trace((true) < 1);
```

The less-than operator converts the value `true` to the numeric value 1 and compares that numeric value to the second operand 1 to return the value `false` (the value 1 is not less than 1).

```
trace(1 < 1); // false
```

You can alter the default left associativity with the parentheses (`()`) operator. You can instruct the compiler to process the less-than operator first by enclosing that operator and its operands in parentheses. The following example uses the parentheses operator to produce a different output using the same numbers as the previous example:

```
trace(3 > (2 < 1)); // true
```

The less-than operator is processed first, which results in a value of `false` because the operand 2 is not less than the operand 1. The value `false` is then passed to the greater-than operator, along with the operand 3. The following code represents this intermediate state:

```
trace(3 > (false));
```

The greater-than operator converts the value `false` to the numeric value 0 and compares that numeric value to the other operand 3 to return `true` (the value 3 is greater than 0).

```
trace(3 > 0); // true
```

The following table lists the operators for ActionScript 3.0 in order of decreasing precedence. Each row of the table contains operators of the same precedence. Each row of operators has higher precedence than the row appearing below it in the table.

Group	Operators
Primary	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Postfix	`x++ x--`
Unary	`++x --x + - ~ ! delete typeof void`
Multiplicative	`* / %`

Group	Operators
Additive	`+ -`
Bitwise shift	`<< >> >>>`
Relational	`< > <= >= as in instanceof is`
Equality	`== != === !==`
Bitwise AND	`&`
Bitwise XOR	`^`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Conditional	`?:`
Assignment	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Comma	`,`

Primary operators

The primary operators include those used for creating Array and Object literals, grouping expressions, calling functions, instantiating class instances, and accessing properties.

All the primary operators, as listed in the following table, have equal precedence. Operators that are part of the E4X specification are indicated by the (E4X) notation.

Operator	Operation performed
`[]`	Initializes an array
`{x:y}`	Initializes an object
`()`	Groups expressions
`f(x)`	Calls a function
`new`	Calls a constructor
`x.y x[y]`	Accesses a property
`<></>`	Initializes an XMLList object (E4X)
`@`	Accesses an attribute (E4X)
`::`	Qualifies a name (E4X)
`..`	Accesses a descendant XML element (E4X)

Postfix operators

The postfix operators take one operator and either increment or decrement the value. Although these operators are unary operators, they are classified separately from the rest of the unary operators because of their higher precedence and special behavior. When a postfix operator is used as part of a larger expression, the expression's value is returned before the postfix operator is processed. For example, the following code shows how the value of the expression `xNum++` is returned before the value is incremented:

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum);   // 1
```

All the postfix operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`++`	Increments (postfix)
`--`	Decrements (postfix)

Unary operators

The unary operators take one operand. The increment (++) and decrement (`--`) operators in this group are *prefix operators*, which means that they appear before the operand in an expression. The prefix operators differ from their postfix counterparts in that the increment or decrement operation is completed before the value of the overall expression is returned. For example, the following code shows how the value of the expression `++xNum` is returned after the value is incremented:

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum);   // 1
```

All the unary operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`++`	Increments (prefix)
`--`	Decrements (prefix)
`+`	Unary +
`-`	Unary - (negation)
`!`	Logical NOT
`~`	Bitwise NOT

Operator	Operation performed
`delete`	Deletes a property
`typeof`	Returns type information
`void`	Returns undefined value

Multiplicative operators

The multiplicative operators take two operands and perform multiplication, division, or modulo calculations.

All the multiplicative operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`*`	Multiplication
`/`	Division
`%`	Modulo

Additive operators

The additive operators take two operands and perform addition or subtraction calculations. All the additive operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`+`	Addition
`-`	Subtraction

Bitwise shift operators

The bitwise shift operators take two operands and shift the bits of the first operand to the extent specified by the second operand. All the bitwise shift operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`<<`	Bitwise left shift
`>>`	Bitwise right shift
`>>>`	Bitwise unsigned right shift

Relational operators

The relational operators take two operands, compare their values, and return a Boolean value. All the relational operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`as`	Checks data type
`in`	Checks for object properties
`instanceof`	Checks prototype chain
`is`	Checks data type

Equality operators

The equality operators take two operands, compare their values, and return a Boolean value. All the equality operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`==`	Equality
`!=`	Inequality
`===`	Strict equality
`!==`	Strict inequality

Bitwise logical operators

The bitwise logical operators take two operands and perform bit-level logical operations. The bitwise logical operators differ in precedence and are listed in the following table in order of decreasing precedence:

Operator	Operation performed
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR

Logical operators

The logical operators take two operands and return a Boolean result. The logical operators differ in precedence and are listed in the following table in order of decreasing precedence:

Operator	Operation performed
`&&`	Logical AND
`||`	Logical OR

Conditional operator

The conditional operator is a ternary operator, which means that it takes three operands. The conditional operator is a shorthand method of applying the `if..else` conditional statement.

Operator	Operation performed
`?:`	Conditional

Assignment operators

The assignment operators take two operands and assign a value to one operand based on the value of the other operand. All the assignment operators, as listed in the following table, have equal precedence:

Operator	Operation performed
`=`	Assignment
`*=`	Multiplication assignment
`/=`	Division assignment
`%=`	Modulo assignment
`+=`	Addition assignment
`-=`	Subtraction assignment
`<<=`	Bitwise left shift assignment
`>>=`	Bitwise right shift assignment
`>>>=`	Bitwise unsigned right shift assignment
`&=`	Bitwise AND assignment
`^=`	Bitwise XOR assignment
`|=`	Bitwise OR assignment

Conditionals

ActionScript 3.0 provides three basic conditional statements that you can use to control program flow.

if..else

The `if..else` conditional statement allows you to test a condition and execute a block of code if that condition exists, or execute an alternative block of code if the condition does not exist. For example, the following code tests whether the value of `x` exceeds 20, generates a `trace()` function if it does, or generates a different `trace()` function if it does not:

```
if (x > 20)
{
  trace("x is > 20");
}
else
{
  trace("x is <= 20");
}
```

If you do not want to execute an alternative block of code, you can use the `if` statement without the `else` statement.

if..else if

You can test for more than one condition using the `if..else if` conditional statement. For example, the following code not only tests whether the value of `x` exceeds 20, but also tests whether the value of `x` is negative:

```
if (x > 20)
{
  trace("x is > 20");
}
else if (x < 0)
{
  trace("x is negative");
}
```

If an `if` or `else` statement is followed by only one statement, the statement does not need to be enclosed in braces. For example, the following code does not use braces.

```
if (x > 0)
  trace("x is positive”);
else if (x < 0)
  trace("x is negative");
else
  trace("x is 0");
```

However, Adobe recommends that you always use braces because unexpected behavior can occur if statements are later added to a conditional statement that lacks braces. For example, in the following code the value of `positiveNums` increases by 1 whether or not the condition evaluates to `true`:

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
  trace("x is positive");
  positiveNums++;

trace(positiveNums); // 1
```

switch

The `switch` statement is useful if you have several execution paths that depend on the same condition expression. It provides functionality similar to a long series of `if..else if` statements, but is somewhat easier to read. Instead of testing a condition for a Boolean value, the `switch` statement evaluates an expression and uses the result to determine which block of code to execute. Blocks of code begin with a `case` statement, and end with a `break` statement. For example, the following `switch` statement prints the day of the week based on the day number returned by the `Date.getDay()` method:

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
  case 0:
    trace("Sunday");
    break;
  case 1:
    trace("Monday");
    break;
  case 2:
    trace("Tuesday");
    break;
  case 3:
    trace("Wednesday");
    break;
  case 4:
    trace("Thursday");
    break;
  case 5:
    trace("Friday");
    break;
  case 6:
    trace("Saturday");
    break;
  default:
    trace("Out of range");
    break;
}
```

Looping

Looping statements allow you to perform a specific block of code repeatedly using a series of values or variables. Adobe recommends that you always enclose the block of code in braces (`{}`). Although you can omit the braces if the block of code contains only one statement, this practice is not recommended for the same reason that it is not recommended for conditionals: it increases the likelihood that statements added later will be inadvertently excluded from the block of code. If you later add a statement that you want to include in the block of code, but forget to add the necessary braces, the statement will not be executed as part of the loop.

for

The `for` loop allows you to iterate through a variable for a specific range of values. You must supply three expressions in a `for` statement: a variable that is set to an initial value, a conditional statement that determines when the looping ends, and an expression that changes the value of the variable with each loop. For example, the following code loops five times. The value of the variable `i` starts at 0 and ends at 4, and the output will be the numbers 0 through 4, each on its own line.

```
var i:int;
for (i = 0; i < 5; i++)
{
  trace(i);
}
```

for..in

The `for..in` loop iterates through the properties of an object, or the elements of an array. For example, you can use a `for..in` loop to iterate through the properties of a generic object (object properties are not kept in any particular order, so properties may appear in a seemingly random order):

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
  trace(i + ": " + myObj[i]);
}
// output:
// x: 20
// y: 30
```

You can also iterate through the elements of an array:

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
```

```
{
  trace(myArray[i]);
}
// output:
// one
// two
// three
```

What you cannot do is iterate through the properties of an object if it is an instance of a user-defined class, unless the class is a dynamic class. Even with instances of dynamic classes, you will be able to iterate only through properties that are added dynamically.

for each..in

The `for each..in` loop iterates through the items of a collection, which can be tags in an XML or XMLList object, the values held by object properties, or the elements of an array. For example, as the following excerpt shows, you can use a `for each..in` loop to iterate through the properties of a generic object, but unlike the `for..in` loop, the iterator variable in a `for each..in` loop contains the value held by the property instead of the name of the property:

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
  trace(num);
}
// output:
// 20
// 30
```

You can iterate through an XML or XMLList object, as the following example shows:

```
var myXML:XML = <users>
                   <fname>Jane</fname>
                   <fname>Susan</fname>
                   <fname>John</fname>
                </users>;

for each (var item in myXML.fname)
{
  trace(item);
}
/* output
Jane
Susan
John
*/
```

You can also iterate through the elements of an array, as this example shows:

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
   trace(item);
}
// output:
// one
// two
// three
```

You cannot iterate through the properties of an object if the object is an instance of a sealed class. Even for instances of dynamic classes, you cannot iterate through any fixed properties, which are properties defined as part of the class definition.

while

The `while` loop is like an `if` statement that repeats as long as the condition is `true`. For example, the following code produces the same output as the `for` loop example:

```
var i:int = 0;
while (i < 5)
{
   trace(i);
   i++;
}
```

One disadvantage of using a `while` loop instead of a `for` loop is that infinite loops are easier to write with `while` loops. The `for` loop example code does not compile if you omit the expression that increments the counter variable, but the `while` loop example does compile if you omit that step. Without the expression that increments `i`, the loop becomes an infinite loop.

do..while

The `do..while` loop is a `while` loop that guarantees that the code block is executed at least once, because the condition is checked after the code block is executed. The following code shows a simple example of a `do..while` loop that generates output even though the condition is not met:

```
var i:int = 5;
do
{
   trace(i);
   i++;
} while (i < 5);
// output: 5
```

Functions

Functions are blocks of code that carry out specific tasks and can be reused in your program. There are two types of functions in ActionScript 3.0: *methods* and *function closures*. Whether a function is a called a method or a function closure depends on the context in which the function is defined. A function is called a method if you define it as part of a class definition or attach it to an instance of an object. A function is called a function closure if it is defined in any other way.

Functions have always been extremely important in ActionScript. In ActionScript 1.0, for example, the `class` keyword did not exist, so "classes" were defined by constructor functions. Although the `class` keyword has since been added to the language, a solid understanding of functions is still important if you want to take full advantage of what the language has to offer. This can be a challenge for programmers who expect ActionScript functions to behave similarly to functions in languages such as C++ or Java. Although basic function definition and invocation should not present a challenge to experienced programmers, some of the more advanced features of ActionScript functions require some explanation.

Basic function concepts

This section discusses basic function definition and invocation techniques.

Calling functions

You call a function by using its identifier followed by the parentheses operator (`()`). You use the parentheses operator to enclose any function parameters you want to send to the function. For example, the `trace()` function, which is a top-level function in the Flash Player API, is used throughout this book:

```
trace("Use trace to help debug your script");
```

If you are calling a function with no parameters, you must use an empty pair of parentheses. For example, you can use the `Math.random()` method, which takes no parameters, to generate a random number:

```
var randomNum:Number = Math.random();
```

Defining your own functions

There are two ways to define a function in ActionScript 3.0: you can use a function statement or a function expression. The technique you choose depends on whether you prefer a more static or dynamic programming style. Define your functions with function statements if you prefer static, or strict mode, programming. Define your functions with function expressions if you have a specific need to do so. Function expressions are more often used in dynamic, or standard mode, programming.

Function statements

Function statements are the preferred technique for defining functions in strict mode. A function statement begins with the `function` keyword, followed by:

- The function name.
- The parameters, in a comma-delimited list enclosed in parentheses.
- The function body—that is, the ActionScript code to be executed when the function is invoked, enclosed in curly braces.

For example, the following code creates a function that defines a parameter and then invokes the function using the string "`hello`" as the parameter value:

```
function traceParameter(aParam:String)
{
  trace(aParam);
}

traceParameter("hello"); // hello
```

Function expressions

The second way to declare a function is to use an assignment statement with a function expression, which is also sometimes called a function literal or an anonymous function. This is a more verbose method that is widely used in earlier versions of ActionScript.

An assignment statement with a function expression begins with the `var` keyword, followed by:

- The function name
- The colon operator (`:`)
- The `Function` class to indicate the data type
- The assignment operator (=)
- The `function` keyword
- The parameters, in a comma-delimited list enclosed in parentheses
- The function body—that is, the ActionScript code to be executed when the function is invoked, enclosed in curly braces

For example, the following code declares the `traceParameter` function using a function expression:

```
var traceParameter:Function = function (aParam:String)
{
  trace(aParam);
};
traceParameter("hello"); // hello
```

Notice that you do not specify a function name, as you do in a function statement. Another important difference between function expressions and function statements is that a function expression is an expression rather than a statement. This means that a function expression cannot stand on its own, as a function statement can. A function expression can be used only as a part of a statement, usually an assignment statement. The following example shows a function expression assigned to an array element:

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
  trace(aParam);
};
traceArray[0]("hello");
```

Choosing between statements and expressions

As a general rule, use a function statement unless specific circumstances call for the use of an expression. Function statements are less verbose, and they provide a more consistent experience between strict mode and standard mode than function expressions.

Function statements are easier to read than assignment statements that contain function expressions. Function statements make your code more concise; they are less confusing than function expressions, which require you to use both the `var` and `function` keywords.

Function statements provide a more consistent experience between the two compiler modes in that you can use dot syntax in both strict and standard mode to invoke a method declared using a function statement. This is not necessarily true for methods declared with a function expression. For example, the following code defines a class named Example with two methods: `methodExpression()`, which is declared with a function expression, and `methodStatement()`, which is declared with a function statement. In strict mode, you cannot use dot syntax to invoke the `methodExpression()` method.

```
class Example
{
  var methodExpression = function() {}
  function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

Function expressions are considered better suited to programming that focuses on run-time, or dynamic, behavior. If you prefer to use strict mode, but also need to call a method declared with a function expression, you can use either of two techniques. First, you can call the method using square brackets (`[]`) instead of the dot (`.`) operator. The following method call succeeds in both strict mode and standard mode:

```
myExample["methodLiteral"]();
```

Second, you can declare the entire class as a dynamic class. Although this allows you to call the method using the dot operator, the downside is that you sacrifice some strict mode functionality for all instances of that class. For example, the compiler does not generate an error if you attempt to access an undefined property on an instance of a dynamic class.

There are some circumstances in which function expressions are useful. One common use of function expressions is for functions that are used only once and then discarded. Another, less common use is for attaching a function to a prototype property. For more information, see "The prototype object" on page 144.

There are two subtle differences between function statements and function expressions that you should take into account when choosing which technique to use. The first difference is that function expressions do not exist independently as objects with regard to memory management and garbage collection. In other words, when you assign a function expression to another object, such as an array element or an object property, you create the only reference to that function expression in your code. If the array or object to which your function expression is attached goes out of scope or is otherwise no longer available, you will no longer have access to the function expression. If the array or object is deleted, the memory that the function expression uses will become eligible for garbage collection, which means that the memory is eligible to be reclaimed and reused for other purposes.

The following example shows that for a function expression, once the property to which the expression is assigned is deleted, the function is no longer available. The class Test is dynamic, which means that you can add a property named `functionExp` that holds a function expression. The `functionExp()` function can be called with the dot operator, but once the `functionExp` property is deleted, the function is no longer accessible.

```
dynamic class Test {}
var myTest:Test = new Test();

// function expression
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp();   // Function expression
delete myTest.functionExp;
myTest.functionExp();   // error
```

If, on the other hand, the function is first defined with a function statement, it exists as its own object, and continues to exist even after you delete the property to which it is attached. The `delete` operator only works on properties of objects, so even a call to delete the function `stateFunc()` itself does not work.

```
dynamic class Test {}
var myTest:Test = new Test();

// function statement
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // Function statement
delete myTest.statement;
delete stateFunc;   // no effect
stateFunc();        // Function statement
myTest.statement(); // error
```

The second difference between function statements and function expressions is that function statements exist throughout the scope in which they are defined, including in statements that appear before the function statement. Function expressions, by contrast, are defined only for subsequent statements. For example, the following code successfully calls the `scopeTest()` function before it is defined:

```
statementTest(); // statementTest

function statementTest():void
{
  trace("statementTest");
}
```

Function expressions are not available before they are defined, so the following code results in a run-time error:

```
expressionTest(); // run-time error

var expressionTest:Function = function ()
{
  trace("expressionTest");
}
```

Returning values from functions

To return a value from your function, use the `return` statement followed by the expression or literal value that you want to return. For example, the following code returns an expression representing the parameter:

```
function doubleNum(baseNum:int):int
{
  return (baseNum * 2);
}
```

Notice that the `return` statement terminates the function, so that any statements below a `return` statement will not be executed, as follows:

```
function doubleNum(baseNum:int):int {
  return (baseNum * 2);
  trace("after return"); // This trace statement will not be executed.
}
```

In strict mode, you must return a value of the appropriate type if you choose to specify a return type. For example, the following code generates an error in strict mode because it does not return a valid value:

```
function doubleNum(baseNum:int):int
{
  trace("after return");
}
```

Nested functions

You can nest functions, which means that functions can be declared within other functions. A nested function is available only within its parent function unless a reference to the function is passed to external code. For example, the following code declares two nested functions inside the `getNameAndVersion()` function:

```
function getNameAndVersion():String
{
  function getVersion():String
  {
    return "9";
  }
  function getProductName():String
  {
    return "Flash Player";
  }
  return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 9
```

When nested functions are passed to external code, they are passed as function closures, which means that the function retains any definitions that are in scope when the function is defined. For more information, see "Function closures" on page 104.

Function parameters

ActionScript 3.0 provides some functionality for function parameters that may seem novel for programmers new to the language. Although the idea of passing parameters by value or reference should be familiar to most programmers, the `arguments` object and the ... (rest) parameter may be new to many of you.

Passing arguments by value or by reference

In many programming languages, it's important to understand the distinction between passing arguments by value or by reference; the distinction can affect the way code is designed.

To be passed by value means that the value of the argument is copied into a local variable for use within the function. To be passed by reference means that only a reference to the argument is passed, instead of the actual value. No copy of the actual argument is made. Instead, a reference to the variable passed as an argument is created and assigned to a local variable for use within the function. As a reference to a variable outside the function, the local variable gives you the ability to change the value of the original variable.

In ActionScript 3.0, all arguments are passed by reference because all values are stored as objects. However, objects that belong to the primitive data types, which includes Boolean, Number, int, uint, and String, have special operators that make them behave as if they were passed by value. For example, the following code creates a function named `passPrimitives()` that defines two parameters named `xParam` and `yParam`, both of type int. These parameters are similar to local variables declared inside the body of the `passPrimitives()` function. When the function is called with the arguments `xValue` and `yValue`, the parameters `xParam` and `yParam` are initialized with references to the int objects represented by `xValue` and `yValue`. Because the arguments are primitives, they behave as if passed by value. Although `xParam` and `yParam` initially contain only references to the `xValue` and `yValue` objects, any changes to the variables within the function body generate new copies of the values in memory.

```
function passPrimitives(xParam:int, yParam:int):void
{
  xParam++;
  yParam++;
  trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);          // 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);          // 10 15
```

Within the `passPrimitives()` function, the values of `xParam` and `yParam` are incremented, but this does not affect the values of `xValue` and `yValue`, as shown in the last `trace` statement. This would be true even if the parameters were named identically to the variables, `xValue` and `yValue`, because the `xValue` and `yValue` inside the function would point to new locations in memory that exist separately from the variables of the same name outside the function.

All other objects—that is, objects that do not belong to the primitive data types—are always passed by reference, which gives you ability to change the value of the original variable. For example, the following code creates an object named `objVar` with two properties, `x` and `y`. The object is passed as an argument to the `passByRef()` function. Because the object is not a primitive type, the object is not only passed by reference, but also stays a reference. This means that changes made to the parameters within the function will affect the object properties outside the function.

```
function passByRef(objParam:Object):void
{
  objParam.x++;
  objParam.y++;
  trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar);         // 11 16
trace(objVar.x, objVar.y); // 11 16
```

The `objParam` parameter references the same object as the global `objVar` variable. As you can see from the `trace` statements in the example, changes to the `x` and `y` properties of the `objParam` object are reflected in `objVar` object.

Default parameter values

New in ActionScript 3.0 is the ability to declare *default parameter values* for a function. If a call to a function with default parameter values omits a parameter with default values, the value specified in the function definition for that parameter is used. All parameters with default values must be placed at the end of the parameter list. The values assigned as default values must be compile-time constants. The existence of a default value for a parameter effectively makes that parameter an *optional parameter*. A parameter without a default value is considered a *required parameter*.

For example, the following code creates a function with three parameters, two of which have default values. When the function is called with only one parameter, the default values for the parameters are used.

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
  trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

The arguments object

When parameters are passed to a function, you can use the `arguments` object to access information about the parameters passed to your function. Some important aspects of the `arguments` object include the following:

- The `arguments` object is an array that includes all the parameters passed to the function.
- The `arguments.length` property reports the number of parameters passed to the function.
- The `arguments.callee` property provides a reference to the function itself, which is useful for recursive calls to function expressions.

NOTE

The `arguments` object is not available if any parameter is named `arguments` or if you use the ... `(rest)` parameter.

ActionScript 3.0 allows function calls to include more parameters than those defined in the function definition, but will generate a compiler error in strict mode if the number of parameters is less than the number of required parameters. You can use the array aspect of the `arguments` object to access any parameter passed to the function, whether or not that parameter is defined in the function definition. The following example uses the `arguments` array along with the `arguments.length` property to trace all the parameters passed to the `traceArgArray()` function:

```
function traceArgArray(x:int):void
{
  for (var i:uint = 0; i < arguments.length; i++)
  {
    trace(arguments[i]);
  }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

The `arguments.callee` property is often used in anonymous functions to create recursion. You can use it to add flexibility to your code. If the name of a recursive function changes over the course of your development cycle, you need not worry about changing the recursive call in your function body if you use `arguments.callee` instead of the function name. The `arguments.callee` property is used in the following function expression to enable recursion:

```
var factorial:Function = function (x:uint)
{
  if(x == 0)
  {
    return 1;
  }
  else
  {
    return (x * arguments.callee(x - 1));
  }
}

trace(factorial(5)); // 120
```

If you use the ... (rest) parameter in your function declaration, the `arguments` object will not be available to you. Instead, you must access the parameters using the parameter names that you declared for them.

You should also be careful to avoid using the string "`arguments`" as a parameter name because it will shadow the `arguments` object. For example, if the function `traceArgArray()` is rewritten so that an `arguments` parameter is added, the references to `arguments` in the function body refer to the parameter rather than the `arguments` object. The following code produces no output:

```
function traceArgArray(x:int, arguments:int):void
{
  for (var i:uint = 0; i < arguments.length; i++)
  {
    trace(arguments[i]);
  }
}

traceArgArray(1, 2, 3);

// no output
```

The `arguments` object in previous versions of ActionScript also contained a property named `caller`, which is a reference to the function that called the current function. The `caller` property is not present in ActionScript 3.0, but if you need a reference to the calling function, you can alter the calling function so that it passes an extra parameter that is a reference to itself.

The ... (rest) parameter

ActionScript 3.0 introduces a new parameter declaration called the ... (rest) parameter. This parameter allows you to specify an array parameter that accepts any number of comma-delimited arguments. The parameter can have any name that is not a reserved word. This parameter declaration must be the last parameter specified. Use of this parameter makes the `arguments` object unavailable. Although the ... (rest) parameter gives you the same functionality as the `arguments` array and `arguments.length` property, it does not provide functionality similar to that provided by `arguments.callee`. You should ensure that you do not need to use `arguments.callee` before using the ... (rest) parameter.

The following example rewrites the `traceArgArray()` function using the ... (rest) parameter instead of the `arguments` object:

```
function traceArgArray(... args):void
{
  for (var i:uint = 0; i < args.length; i++)
  {
    trace(args[i]);
  }
}

traceArgArray(1, 2, 3);
```

```
// output:
// 1
// 2
// 3
```

The ... (rest) parameter can also be used with other parameters, as long as it is the last parameter listed. The following example modifies the `traceArgArray()` function so that its first parameter, `x`, is of type int, and the second parameter uses the ... (rest) parameter. The output skips the first value because the first parameter is no longer part of the array created by the ... (rest) parameter.

```
function traceArgArray(x: int, ... args)
{
  for (var i:uint = 0; i < args.length; i++)
  {
    trace(args[i]);
  }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

Functions as objects

Functions in ActionScript 3.0 are objects. When you create a function, you are creating an object that can not only be passed as a parameter to another function, but also have properties and methods attached to it.

Functions passed as arguments to another function are passed by reference and not by value. When you pass a function as an argument, you use only the identifier and not the parentheses operator that you use to call the method. For example, the following code passes a function named `clickListener()` as an argument to the `addEventListener()` method:

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

The `Array.sort()` method also defines a parameter that accepts a function. For an example of a custom sort function that is used as an argument to the `Array.sort()` function, see "Sorting an array" on page 231.

Although it may seem strange to programmers new to ActionScript, functions can have properties and methods, just as any other object can. In fact, every function has a read-only property named `length` that stores the number of parameters defined for the function. This is different from the `arguments.length` property, which reports the number of arguments sent to the function. Recall that in ActionScript, the number of arguments sent to a function can exceed the number of parameters defined for that function. The following example, which compiles only in standard mode because strict mode requires an exact match between the number of arguments passed and the number of parameters defined, shows the difference between the two properties:

```
function traceLength (x:uint, y:uint):void
{
  trace("arguments received: " + arguments.length);
  trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

You can define your own function properties by defining them outside your function body. Function properties can serve as quasi-static properties that allow you to save the state of a variable related to the function. For example, you may want to track the number of times a particular function is called. Such functionality could be useful if you are writing a game and want to track the number of times a user uses a specific command, although you could also use a static class property for this. The following code creates a function property outside the function declaration and increments the property each time the function is called:

```
someFunction.counter = 0;

function someFunction():void
{
  someFunction.counter++;
}

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

Function scope

A function's scope determines not only where in a program that function can be called, but also what definitions the function can access. The same scope rules that apply to variable identifiers apply to function identifiers. A function declared in the global scope is available throughout your code. For example, ActionScript 3.0 contains global functions, such as `isNaN()` and `parseInt()`, that are available anywhere in your code. A nested function—a function declared within another function—can be used anywhere in the function in which it was declared.

The scope chain

Any time a function begins execution, a number of objects and properties are created. First, a special object called an *activation object* is created that stores the parameters and any local variables or functions declared in the function body. You cannot access the activation object directly because it is an internal mechanism. Second, a *scope chain* is created that contains an ordered list of objects that Flash Player checks for identifier declarations. Every function that executes has a scope chain that is stored in an internal property. For a nested function, the scope chain starts with its own activation object, followed by its parent function's activation object. The chain continues in this manner until it reaches the global object. The global object is created when an ActionScript program begins, and contains all global variables and functions.

Function closures

A *function closure* is an object that contains a snapshot of a function and its *lexical environment*. A function's lexical environment includes all the variables, properties, methods, and objects in the function's scope chain, along with their values. Function closures are created any time a function is executed apart from an object or a class. The fact that function closures retain the scope in which they were defined creates interesting results when a function is passed as an argument or a return value into a different scope.

For example, the following code creates two functions: `foo()`, which returns a nested function named `rectArea()` that calculates the area of a rectangle, and `bar()`, which calls `foo()` and stores the returned function closure in a variable named `myProduct`. Even though the `bar()` function defines its own local variable `x` (with a value of 2), when the function closure `myProduct()` is called, it retains the variable `x` (with a value of 40) defined in function `foo()`. The `bar()` function therefore returns the value `160` instead of `8`.

```
function foo():Function
{
  var x:int = 40;
  function rectArea(y:int):int // function closure defined
  {
    return x * y
  }
  return rectArea;
}
function bar():void
{
  var x:int = 2;
  var y:int = 4;
  var myProduct:Function = foo();
  trace( myProduct(4) ); // function closure called
}
bar(); // 160
```

Methods behave similarly in that they also retain information about the lexical environment in which they were created. This characteristic is most noticeable when a method is extracted from its instance, which creates a bound method. The main difference between a function closure and a bound method is that the value of the `this` keyword in a bound method always refers to the instance to which it was originally attached, whereas in a function closure the value of the `this` keyword can change. For more information, see "Bound methods" on page 121.

CHAPTER 4

Object-Oriented Programming in ActionScript

4

This chapter assumes a basic understanding of object-oriented programming (OOP) principles such as abstraction, encapsulation, inheritance, and polymorphism. The chapter focuses on how to apply these principles using ActionScript 3.0.

Because of ActionScript's roots as a scripting language, ActionScript 3.0 OOP support is optional. This affords programmers flexibility in choosing the best approach for projects of varying scope and complexity. For small tasks, you may find that using ActionScript with a procedural programming paradigm is all you need. For larger projects, applying OOP principles can make your code easier to understand, maintain, and extend.

Contents

Classes

A class is an abstract representation of an object. A class stores information about the types of data that an object can hold and the behaviors that an object can exhibit. The usefulness of such an abstraction may not be apparent when you write small scripts that contain only a few objects interacting with one another. As the scope of a program grows, however, and the number of objects that must be managed increases, you may find that classes allow you to better control how objects are created and how they interact with one another.

As far back as ActionScript 1.0, ActionScript programmers could use Function objects to create constructs that resembled classes. ActionScript 2.0 added formal support for classes with keywords such as `class` and `extends`. ActionScript 3.0 not only continues to support the keywords introduced in ActionScript 2.0, but also adds some new capabilities, such as enhanced access control with the `protected` and `internal` attributes, and better control over inheritance with the `final` and `override` keywords.

If you have ever created classes in programming languages like Java, C++, or C#, you will find that ActionScript provides a familiar experience. ActionScript shares many of the same keywords and attribute names, such as `class`, `extends`, and `public`, all of which are discussed in the following sections.

NOTE

In this chapter, the term *property* means any member of an object or class, including variables, constants, and methods. In addition, although the terms *class* and *static* are often used interchangeably, in this chapter these terms are distinct. For example, in this chapter the phrase *class properties* refers to all the members of a class, rather than only the static members.

Class definitions

ActionScript 3.0 class definitions use syntax that is similar to that used in ActionScript 2.0 class definitions. Proper syntax for a class definition calls for the `class` keyword followed by the class name. The class body, which is enclosed by curly braces(`{}`), follows the class name. For example, the following code creates a class named Shape that contains one variable, named `visible`:

```
public class Shape
{
  var visible:Boolean = true;
}
```

One significant syntax change involves class definitions that are inside a package. In ActionScript 2.0, if a class is inside a package, the package name must be included in the class declaration. In ActionScript 3.0, which introduces the `package` statement, the package name must be included in the package declaration instead of in the class declaration. For example, the following class declarations show how the BitmapData class, which is part of the flash.display package, is defined in ActionScript 2.0 and ActionScript 3.0:

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

Class attributes

ActionScript 3.0 allows you to modify class definitions using one of the following four attributes:

Attribute	Definition
`dynamic`	Allow properties to be added to instances at run time.
`final`	Must not be extended by another class.
`internal` (default)	Visible to references inside the current package.
`public`	Visible to references everywhere.

For each of these attributes, except for `internal`, you must explicitly include the attribute to get the associated behavior. For example, if you do not include the `dynamic` attribute when defining a class, you will not be able to add properties to a class instance at run time. You explicitly assign an attribute by placing it at the beginning of the class definition, as the following code demonstrates:

```
dynamic class Shape {}
```

Notice that the list does not include an attribute named `abstract`. This is because abstract classes are not supported in ActionScript 3.0. Notice also that the list does not include attributes named `private` and `protected`. These attributes have meaning only inside a class definition, and cannot be applied to classes themselves. If you do not want a class to be publicly visible outside a package, place the class inside a package and mark the class with the `internal` attribute. Alternatively, you can omit both the `internal` and `public` attributes, and the compiler will automatically add the `internal` attribute for you. If you do not want a class to be visible outside the source file in which it is defined, place the class at the bottom of your source file below the closing curly brace of the package definition.

Class body

The class body, which is enclosed by curly braces, is used to define the variables, constants, and methods of your class. The following example shows the declaration for the Accessibility class in the Flash Player API:

```
public final class Accessibility
{
  public static function get active():Boolean;
  public static function updateProperties():void;
}
```

You can also define a namespace inside a class body. The following example shows how a namespace can be defined within a class body and used as an attribute of a method in that class:

```
public class SampleClass
{
  public namespace sampleNamespace;
  sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 allows you to include not only definitions in a class body, but also statements. Statements that are inside a class body, but outside a method definition, are executed exactly once—when the class definition is first encountered and the associated class object is created. The following example includes a call to an external function, `hello()`, and a `trace` statement that outputs a confirmation message when the class is defined:

```
function hello():String
{
  trace ("hola");
}
class SampleClass
{
  hello();
  trace("class created");
}
// output when class is created
hola
class created
```

In contrast to previous versions of ActionScript, in ActionScript 3.0 it is permissible to define a static property and an instance property with the same name in the same class body. For example, the following code declares a static variable named `message` and an instance variable of the same name:

```
class StaticTest
{
  static var message:String = "static variable";
  var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // static variable
trace(myST.message);       // instance variable
```

Class property attributes

In discussions of the ActionScript object model, the term *property* means anything that can be a member of a class, including variables, constants, and methods. This differs from the way the term is used in the *ActionScript 3.0 Language Reference*, where the term is used more narrowly and includes only class members that are variables or are defined by a getter or setter method. In ActionScript 3.0, there is a set of attributes that can be used with any property of a class. The following table lists this set of attributes.

Attribute	Definition
`internal` (default)	Visible to references inside the same package.
`private`	Visible to references in the same class.
`protected`	Visible to references in the same class and derived classes.
`public`	Visible to references everywhere.
`static`	Specifies that a property belongs to the class, as opposed to instances of the class.
`UserDefinedNamespace`	Custom namespace name defined by user.

Access control namespace attributes

ActionScript 3.0 provides four special attributes that control access to properties defined inside a class: `public`, `private`, `protected`, and `internal`.

The `public` attribute makes a property visible anywhere in your script. For example, to make a method available to code outside its package, you must declare the method with the `public` attribute. This is true for any property, whether it is declared using the `var`, `const`, or `function` keywords.

The `private` attribute makes a property visible only to callers within the property's defining class. This behavior differs from that of the `private` attribute in ActionScript 2.0, which allowed a subclass to access a private property in a superclass. Another significant change in behavior has to do with run-time access. In ActionScript 2.0, the `private` keyword prohibited access only at compile time, and was easily circumvented at run time. In ActionScript 3.0, this is no longer true. Properties that are marked as `private` are unavailable at both compile time and run time.

For example, the following code creates a simple class named PrivateExample with one private variable, and then attempts to access the private variable from outside the class. In ActionScript 2.0, compile-time access was prohibited, but the prohibition was easily circumvented by using the property access operator (`[]`), which does the property lookup at run time rather than at compile time.

```
class PrivateExample
{
  private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);    // compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in
  ActionScript 3.0, this is a run-time error.
```

In ActionScript 3.0, an attempt to access a private property using the dot operator (`myExample.privVar`) is a compile-time error if you are using strict mode. Otherwise, the error is reported at run time, just as it is when you use the property access operator (`myExample["privVar"]`).

The following table summarizes the results of attempting to access a private property that belongs to a sealed (not dynamic) class:

	Strict mode	Standard mode
dot operator (.)	compile-time error	run-time error
bracket operator ([])	run-time error	run-time error

In classes declared with the `dynamic` attribute, attempts to access a private variable will not result in a run-time error. Instead, the variable is simply not visible, so Flash Player returns the value `undefined`. A compile-time error occurs, however, if you use the dot operator in strict mode. The following example is the same as the previous example, except that the PrivateExample class is declared as a dynamic class:

```
dynamic class PrivateExample
{
  private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);    // compile-time error in strict mode
trace(myExample["privVar"]); // output: undefined
```

Dynamic classes generally return the value `undefined` instead of generating an error when code external to a class attempts to access a private property. The following table shows that an error is generated only when the dot operator is used to access a private property in strict mode:

	Strict mode	**Standard mode**
dot operator (`.`)	compile-time error	`undefined`
bracket operator (`[]`)	`undefined`	`undefined`

The `protected` attribute, which is new for ActionScript 3.0, makes a property visible to callers within its own class or in a subclass. In other words, a protected property is available within its own class or to classes that lie anywhere below it in the inheritance hierarchy. This is true whether the subclass is in the same package or in a different package.

For those familiar with ActionScript 2.0, this functionality is similar to the `private` attribute in ActionScript 2.0. The ActionScript 3.0 `protected` attribute is also similar to the `protected` attribute in Java, but differs in that the Java version also permits access to callers within the same package. The `protected` attribute is useful when you have a variable or method that your subclasses needs but that you want to hide from code that is outside the inheritance chain.

The `internal` attribute, which is new for ActionScript 3.0, makes a property visible to callers within its own package. This is the default attribute for code inside a package, and it applies to any property that does not have any of the following attributes:

- `public`
- `private`
- `protected`
- a user-defined namespace

The `internal` attribute is similar to the default access control in Java, although in Java there is no explicit name for this level of access, and it can be achieved only through the omission of any other access modifier. The `internal` attribute is available in ActionScript 3.0 to give you the option of explicitly signifying your intent to make a property visible only to callers within its own package.

static attribute

The `static` attribute, which can be used with properties declared with the `var`, `const`, or `function` keywords, allows you to attach a property to the class rather than to instances of the class. Code external to the class must call static properties using the class name instead of an instance name.

Static properties are not inherited by subclasses, but the properties are part of a subclass's scope chain. This means that within the body of a subclass, a static variable or method can be used without referencing the class in which it was defined. For more information, see "Static properties not inherited" on page 136.

User-defined namespace attributes

As an alternative to the predefined access control attributes, you can create a custom namespace for use as an attribute. Only one namespace attribute can be used per definition, and you cannot use a namespace attribute in combination with any of the access control attributes (`public`, `private`, `protected`, `internal`). For more information about using namespaces, see "Namespaces" on page 43.

Variables

Variables can be declared with either the `var` or `const` keywords. Variables declared with the `var` keyword can have their values changed multiple times throughout the execution of a script. Variables declared with the `const` keyword are called constants, and can have values assigned to them only once. An attempt to assign a new value to an initialized constant results in an error. For more information, see "Constants" on page 76.

Static variables

Static variables are declared using a combination of the `static` keyword and either the `var` and `const` statements. Static variables, which are attached to a class rather than an instance of a class, are useful for storing and sharing information that applies to an entire class of objects. For example, a static variable is appropriate if you want to keep a tally of the number of times a class is instantiated or if you want to store the maximum number of class instances that are allowed.

The following example creates a `totalCount` variable to track the number of class instantiations and a `MAX_NUM` constant to store the maximum number of instantiations. The `totalCount` and `MAX_NUM` variables are static because they contain values that apply to the class as a whole rather than to a particular instance.

```
class StaticVars
{
  public static var totalCount:int = 0;
  public static const MAX_NUM:uint = 16;
}
```

Code that is external to the StaticVars class and any of its subclasses can reference the `totalCount` and `MAX_NUM` properties only through the class itself. For example, the following code works:

```
trace(StaticVars.totalCount); // 0
trace(StaticVars.MAX_NUM); // 16
```

You cannot access static variables through an instance of the class, so the following code returns errors:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Variables that are declared with both the `static` and `const` keywords must be initialized at the same time as you declare the constant, as the StaticVars class does for `MAX_NUM`. You cannot assign a value to `MAX_NUM` inside the constructor or an instance method. The following code will generate an error because it is not a valid way to initialize a static constant:

```
// !! Error to initialize static constant this way
class StaticVars2
{
  public static const UNIQUESORT:uint;
  function initializeStatic():void
  {
    UNIQUESORT = 16;
  }
}
```

Instance variables

Instance variables include properties declared with the `var` and `const` keywords, but without the `static` keyword. Instance variables, which are attached to class instances rather than to an entire class, are useful for storing values that are specific to an instance. For example, the Array class has an instance property named `length` that stores the number of array elements that a particular instance of the Array class holds.

Instance variables, whether declared as `var` or `const`, cannot be overridden in a subclass. You can, however, achieve functionality that is similar to overriding variables by overriding getter and setter methods. For more information, see "Get and set accessor methods" on page 120.

Methods

Methods are functions that are part of a class definition. Once an instance of the class is created, a method is bound to that instance. Unlike a function declared outside a class, a method cannot be used apart from the instance to which it is attached.

Methods are defined using the `function` keyword. You can use a function statement, such as the following:

```
public function sampleFunction():String {}
```

Or you can use a variable to which you assign a function expression, as follows:

```
public var sampleFunction:Function = function () {}
```

In most cases you will want to use a function statement instead of a function expression for the following reasons:

- Function statements are more concise and easier to read.
- Function statements allow you to use the `override` and `final` keywords. For more information, see "Overriding methods" on page 134.
- Function statements create a stronger bond between the identifier—that is, the name of the function—and the code within the method body. Because the value of a variable can be changed with an assignment statement, the connection between a variable and its function expression can be severed at any time. Although you can work around this issue by declaring the variable with `const` instead of `var`, such a technique is not considered a best practice because it makes the code hard to read and prevents the use of the `override` and `final` keywords.

One case in which you must use a function expression is when you choose to attach a function to the prototype object. For more information, see "The prototype object" on page 144.

Constructor methods

Constructor methods, sometimes simply called constructors, are functions that share the same name as the class in which they are defined. Any code that you include in a constructor method is executed whenever an instance of the class is created with the `new` keyword. For example, the following code defines a simple class named Example that contains a single property named `status`. The initial value of the `status` variable is set inside the constructor function.

```
class Example
{
  public var status:String;
  public function Example()
  {
    status = "initialized";
  }
}

var myExample:Example = new Example();
trace (myExample.status); // output: initialized
```

Constructor methods can only be public, but the use of the `public` attribute is optional. You cannot use any of the other access control specifiers, including `private`, `protected`, or `internal`, on a constructor. You also cannot use a user-defined namespace with a constructor method.

A constructor can make an explicit call to the constructor of its direct superclass using the `super()` statement. If the superclass constructor is not explicitly called, the compiler automatically inserts a call before the first statement in the constructor body. You can also call methods of the superclass using the `super` prefix as a reference to the superclass. If you decide to use both `super()` and `super` in the same constructor body, be sure to call `super()` first. Otherwise, the `super` reference will not behave as expected. The `super()` constructor should also be called before any `throw` or `return` statement.

The following example demonstrates what happens if you attempt to use the `super` reference before calling the `super()` constructor. A new class, ExampleEx, extends the Example class. The ExampleEx constructor attempts to access the status variable defined in its superclass, but does so before calling `super()`. The `trace()` statement inside the ExampleEx constructor produces the value `null` because the `status` variable is not available until the `super()` constructor executes.

```
class ExampleEx extends Example
{
  public function ExampleEx()
  {
    trace(super.status);
    super();
  }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Although it is legal to use the `return` statement inside a constructor, it is not permissible to return a value. In other words, `return` statements must not have associated expressions or values. Accordingly, constructor methods are not allowed to return values, which means that no return type may be specified.

If you do not define a constructor method in your class, the compiler will automatically create an empty constructor for you. If your class extends another class class, the compiler will include a `super()` call in the constructor it generates.

Static methods

Static methods, also called class methods, are methods that are declared with the `static` keyword. Static methods, which are attached to a class rather than to an instance of a class, are useful for encapsulating functionality that affects something other than the state of an individual instance. Because static methods are attached to a class as a whole, static methods can be accessed only through a class and not through an instance of the class.

Static methods are useful for encapsulating functionality that is not limited to affecting the state of class instances. In other words, a method should be static if it provides functionality that does not directly affect the value of a class instance. For example, the Date class has a static method named `parse()`, which takes a string and converts it to a number. The method is static because it does not affect an individual instance of the class. Instead, the `parse()` method takes a string that represents a date value, parses the string, and returns a number in a format compatible with the internal representation of a Date object. This method is not an instance method because it does not make sense to apply the method to an instance of the Date class.

Contrast the static `parse()` method with one of the instance methods of the Date class, such as `getMonth()`. The `getMonth()` method is an instance method because it operates directly on the value of an instance by retrieving a specific component, the month, of a Date instance.

Because static methods are not bound to individual instances, you cannot use the keywords `this` or `super` within the body of a static method. Both the `this` reference and the `super` reference have meaning only within the context of an instance method.

In contrast with some other class-based programming languages, static methods in ActionScript 3.0 are not inherited. For more information, see "Static properties not inherited" on page 136.

Instance methods

Instance methods are methods that are declared without the `static` keyword. Instance methods, which are attached to instances of a class instead of the class as a whole, are useful for implementing functionality that affects individual instances of a class. For example, the Array class contains an instance method named `sort()`, which operates directly on Array instances.

Within the body of an instance method, both static and instance variables are in scope, which means that variables defined in the same class can be referenced using a simple identifier. For example, the following class, CustomArray, extends the Array class. The CustomArray class defines a static variable named `arrayCountTotal` to track the total number of class instances, an instance variable named `arrayNumber` that tracks the order in which the instances were created, and an instance method named `getPosition()` that returns the values of these variables.

```
public class CustomArray extends Array
{
  public static var arrayCountTotal:int = 0;
  public var arrayNumber:int;

  public function CustomArray()
  {
    arrayNumber = ++arrayCountTotal;
  }

  public function getArrayPosition():String
  {
     return ("Array " + arrayNumber + " of " + arrayCountTotal);
  }
}
```

Although code external to the class must refer to the `arrayCountTotal` static variable through the class object using `CustomArray.arrayCountTotal`, code that resides inside the body of the `getPosition()` method can refer directly to the static `arrayCountTotal` variable. This is true even for static variables in superclasses. Though static properties are not inherited in ActionScript 3.0, static properties in superclasses are in scope. For example, the Array class has a few static variables, one of which is a constant named `DESCENDING`. Code that resides in an Array subclass can refer to the static constant `DESCENDING` using a simple identifier.

```
public class CustomArray extends Array
{
  public function testStatic():void
  {
    trace(DESCENDING); // output: 2
  }
}
```

The value of the `this` reference within the body of an instance method is a reference to the instance to which the method is attached. The following code demonstrates that the `this` reference points to the instance that contains the method:

```
class ThisTest
{
  function thisValue():ThisTest
```

```
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // true
```

Inheritance of instance methods can be controlled with the keywords `override` and `final`. You can use the `override` attribute to redefine an inherited method, and the `final` attribute to prevent subclasses from overriding a method. For more information, see "Overriding methods" on page 134.

Get and set accessor methods

Get and set accessor functions, also called *getters* and *setters*, allow you to adhere to the programming principles of information hiding and encapsulation while providing an easy-to-use programming interface for the classes that you create. Get and set functions allow you to keep your class properties private to the class, but allow users of your class to access those properties as if they were accessing a class variable instead of calling a class method.

The advantage of this approach is that it allows you to avoid the traditional accessor functions with unwieldy names, such as `getPropertyName()` and `setPropertyName()`. Another advantage of getters and setters is that you can avoid having two public-facing functions for each property that allows both read and write access.

The following example class, named GetSet, includes get and set accessor functions named `publicAccess()` that provide access to the private variable named `privateProperty`:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

If you attempt to access the property `privateProperty` directly, an error will result, as follows:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Instead, a user of the GetSet class will use something that appears to be a property named `publicAccess`, but that is really a pair of get and set accessor functions that operate on the private property named `privateProperty`. The following example instantiates the GetSet class, and then sets the value of the `privateProperty` using the public accessor named `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // hello
```

Getter and setter functions also make it possible to override properties that are inherited from a superclass, something that is not possible when you use regular class member variables. Class member variables that are declared using the `var` keyword cannot be overridden in a subclass. Properties that are created using getter and setter functions, however, do not have this restriction. You can use the `override` attribute on getter and setter functions that are inherited from a superclass.

Bound methods

A bound method, sometimes called a method closure, is simply a method that is extracted from its instance. Examples of bound methods include methods that are passed as arguments to a function or returned as values from a function. New in ActionScript 3.0, a bound method is similar to a function closure in that it retains its lexical environment even when extracted from its instance. The key difference, however, between a bound method and a function closure is that the `this` reference for a bound method remains linked, or bound, to the instance that implements the method. In other words, the `this` reference in a bound method always points to the original object that implemented the method. For function closures, the `this` reference is generic, which means that it points to whatever object the function is associated with at the time it is invoked.

Understanding bound methods is important if you use the `this` keyword. Recall that the `this` keyword provides a reference to a method's parent object. Most ActionScript programmers expect that the `this` keyword always refers to the object or class that contains the definition of a method. Without method binding, however, this would not always be true. In previous versions of ActionScript, for example, the `this` reference did not always refer to the instance that implemented the method. When methods are extracted from an instance in ActionScript 2.0, not only is the `this` reference not bound to the original instance, but also the member variables and methods of the instance's class are not available. This is not a problem in ActionScript 3.0 because bound methods are automatically created when you pass a method as a parameter. Bound methods ensure that the `this` keyword always references the object or class in which a method is defined.

The following code defines a class named ThisTest, which contains a method named `foo()` that defines the bound method, and a method named `bar()` that returns the bound method. Code external to the class creates an instance of the ThisTest class, calls the `bar()` method, and stores the return value in a variable named `myFunc`.

```
class ThisTest
{
  private var num:Number = 3;
  function foo():void // bound method defined
  {
    trace("foo's this: " + this);
    trace("num: " + num);
  }
  function bar():Function
  {
    return foo; // bound method returned
  }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
num: 3 */
```

The last two lines of code show that the `this` reference in the bound method `foo()` still points to an instance of ThisTest class, even though the `this` reference in the line just before it points to the global object. Moreover, the bound method stored in the `myFunc` variable still has access to the member variables of the ThisTest class. If this same code is run in ActionScript 2.0, the `this` references would match and the `num` variable would be `undefined`.

One area where the addition of bound methods is most noticeable is with event handlers, because the `addEventListener()` method requires that you pass a function or method as an argument. For more information, see "Listener function defined as a class method" on page 359.

Enumerations with classes

Enumerations are custom data types that you create to encapsulate a small set of values. ActionScript 3.0 does not support a specific enumeration facility, unlike C++ with its `enum` keyword or Java with its Enumeration interface. You can, however, create enumerations using classes and static constants. For example, the PrintJob class in the Flash Player API uses an enumeration named PrintJobOrientation to store the set of values comprising `"landscape"` and `"portrait"`, as shown in the following code:

```
public final class PrintJobOrientation
{
  public static const LANDSCAPE:String = "landscape";
  public static const PORTRAIT:String = "portrait";
}
```

By convention, an enumeration class is declared with the `final` attribute because there is no need to extend the class. The class comprises only static members, which means that you do not create instances of the class. Instead, you access the enumeration values directly through the class object, as shown in the following code excerpt:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
  if(pj.orientation == PrintJobOrientation.PORTRAIT)
  {
    ...
  }
  ...
}
```

All of the enumeration classes in the Flash Player API contain only variables of type String, int, or uint. The advantage of using enumerations instead of literal string or number values is that typographical mistakes are easier to find with enumerations. If you mistype the name of an enumeration, the ActionScript compiler generates an error. If you use literal values, the compiler does not complain if you spell a word incorrectly or use the wrong number. In the previous example, the compiler generates an error if the name of the enumeration constant is incorrect, as the following excerpt shows:

```
  if(pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

However, the compiler does not generate an error if you misspell a string literal value, as follows:

```
  if(pj.orientation == "portrai") // no compiler error
```

A second technique for creating enumerations also involves creating a separate class with static properties for the enumeration. This technique differs, however, in that each of the static properties contains an instance of the class instead of a string or integer value. For example, the following code creates an enumeration class for the days of the week:

```
public final class Day
{
  public static const MONDAY:Day = new Day();
  public static const TUESDAY:Day = new Day();
  public static const WEDNESDAY:Day = new Day();
  public static const THURSDAY:Day = new Day();
  public static const FRIDAY:Day = new Day();
  public static const SATURDAY:Day = new Day();
  public static const SUNDAY:Day = new Day();
}
```

This technique is not used by the Flash Player API, but is used by many developers who prefer the improved type checking that the technique provides. For example, a method that returns an enumeration value can restrict the return value to the enumeration data type. The following code shows not only a function that returns a day of the week, but also a function call that uses the enumeration type as a type annotation:

```
function getDay():Day
{
  var date:Date = new Date();
  var retDay:Day;
  switch (date.day)
  {
    case 0:
      retDay = Day.MONDAY;
      break;
    case 1:
      retDay = Day.TUESDAY;
      break;
    case 2:
      retDay = Day.WEDNESDAY;
      break;
    case 3:
      retDay = Day.THURSDAY;
      break;
    case 4:
      retDay = Day.FRIDAY;
      break;
    case 5:
      retDay = Day.SATURDAY;
      break;
    case 6:
      retDay = Day.SUNDAY;
      break;
    }
    return retDay;
}

var dayOfWeek:Day = getDay();
```

You can also enhance the Day class so that it associates an integer with each day of the week, and provides a `toString()` method that returns a string representation of the day. You might want to enhance the Day class in this manner as an exercise.

Embedded asset classes

ActionScript 3.0 uses special classes, called embedded asset classes, to represent embedded assets. An *embedded asset* is an asset, such as a sound, image, or font, that is included in a SWF file at compile time. Embedding an asset instead of loading it dynamically ensures that it will be available at run time, but at the cost of increased SWF file size.

Using embedded asset classes in Flex

To embed an asset in ActionScript code, use the `[Embed]` metadata tag. Place the asset in the main source folder or another folder that is in your project's build path. When the Flex compiler encounters an Embed metadata tag, it creates the embedded asset class for you. You can access the class through a variable of data type Class that you declare immediately following the `[Embed]` metadata tag. For example, the following code embeds a sound named sound1.mp3 and uses a variable named `soundCls` to store a reference to the embedded asset class associated with that sound. The example then creates an instance of the embedded asset class and calls the `play()` method on that instance:

<<below code sample was double conditioned to FlexOnly+No Trans. I had to emove FlexOnly for Flex 2.0.1 loc prodeuction. Since this book is in Flex 2.0.1 location and should not be used for Flash-MN>>

```
package
{
    import flash.display.Sprite;
    import flash.media.SoundChannel;
    import mx.core.SoundAsset;

    public class SoundAssetExample extends Sprite
    {
        [Embed(source="sound1.mp3")]
        public var soundCls:Class;

        public function SoundAssetExample()
        {
            var mySound:SoundAsset = new soundCls() as SoundAsset;
            var sndChannel:SoundChannel = mySound.play();
        }
    }
}
```

TIP

To use the `[Embed]` metadata tag in an Adobe Flex Builder 2 ActionScript project, you must import any necessary classes from the Flex framework. For example, to embed sounds, you must import the mx.core.SoundAsset class. To use the Flex framework, include the file framework.swc in your ActionScript build path. This will increase the size of your SWF file.

Alternatively, you can embed an asset with the `@Embed()` directive in an MXML tag definition. For more information, see "About embedding assets" in the *Flex 2 Developer's Guide*.

Interfaces

An interface is a collection of method declarations that allows unrelated objects to communicate with one another. For example, the Flash Player API defines the IEventDispatcher interface, which contains method declarations that a class can use to handle event objects. The IEventDispatcher interface establishes a standard way for objects to pass event objects to one another. The following code shows the definition of the IEventDispatcher interface:

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
        useCapture:Boolean=false, priority:int=0,
        useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
        useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

Interfaces are based on the distinction between a method's interface and its implementation. A method's interface includes all the information necessary to invoke that method, including the name of the method, all of its parameters, and its return type. A method's implementation includes not only the interface information, but also the executable statements that carry out the method's behavior. An interface definition contains only method interfaces, and any class that implements the interface is responsible for defining the method implementations.

In the Flash Player API, the EventDispatcher class implements the IEventDispatcher interface by defining all of the IEventDispatcher interface methods and adding method bodies to each of the methods. The following code is an excerpt from the EventDispatcher class definition:

```
public class EventDispatcher implements IEventDispatcher
{
  function dispatchEvent(event:Event):Boolean
  {
    /* implementation statements */
  }

  ...
}
```

The IEventDispatcher interface serves as a protocol that EventDispatcher instances use to process event objects and pass them to other objects that have also implemented the IEventDispatcher interface.

Another way to describe an interface is to say that it defines a data type just as a class does. Accordingly, an interface can be used as a type annotation, just as a class can. As a data type, an interface can also be used with operators, such as the `is` and `as` operators, that require a data type. Unlike a class, however, an interface cannot be instantiated. This distinction has led many programmers to think of interfaces as abstract data types and classes as concrete data types.

Defining an interface

The structure of an interface definition is similar to that of a class definition, except that an interface can contain only methods with no method bodies. Interfaces cannot include variables or constants, but can include getters and setters. To define an interface, use the `interface` keyword. For example, the following interface, IExternalizable, is part of the flash.utils package in the Flash Player API. The IExternalizable interface defines a protocol for serializing an object, which means converting an object into a format suitable for storage on a device or for transport across a network.

```
public interface IExternalizable
{
  function writeExternal(output:IDataOutput):void;
  function readExternal(input:IDataInput):void;
}
```

Note that the IExternalizable interface is declared with the `public` access control modifier. Interface definitions may only be modified by the `public` and `internal` access control specifiers. The method declarations inside an interface definition cannot have any access control specifiers.

The Flash Player API follows a convention in which interface names begin with an uppercase `I`, but you can use any legal identifier as an interface name. Interface definitions are often placed at the top level of a package. Interface definitions cannot be placed inside a class definition or inside another interface definition.

Interfaces can extend one or more other interfaces. For example, the following interface, IExample, extends the IExternalizable interface:

```
public interface IExample extends IExternalizable
{
  function extra():void;
}
```

Any class that implements the IExample interface must include implementations not only for the `extra()` method, but also for the `writeExternal()` and `readExternal()` methods inherited from the IExternalizable interface.

Implementing an interface in a class

A class is the only ActionScript 3.0 language element that can implement an interface. Use the `implements` keyword in a class declaration to implement one or more interfaces. The following example defines two interfaces, IAlpha and IBeta, and a class, Alpha, that implements them both:

```
interface IAlpha
{
  function foo(str:String):String;
}

interface IBeta
{
  function bar():void;
}

class Alpha implements IAlpha, IBeta
{
  public function foo(param:String):String {}
  public function bar():void {}
}
```

In a class that implements an interface, implemented methods must do the following:

- Use the `public` access control identifier.
- Use the same name as the interface method.
- Have the same number of parameters, each with data types that match the interface method parameter data types.
- Use the same return type.

You do have some flexibility, however, in how you name the parameters of methods that you implement. Although the number of parameters and the data type of each parameter in the implemented method must match that of the interface method, the parameter names do not need to match. For example, in the previous example the parameter of the `Alpha.foo()` method is named `param`:

```
public function foo(param:String):String {}
```

But the parameter is named `str` in the `IAlpha.foo()` interface method:

```
function foo(str:String):String;
```

You also have some flexibility with default parameter values. An interface definition can include function declarations with default parameter values. A method that implements such a function declaration must have a default parameter value that is a member of the same data type as the value specified in the interface definition, but the actual value does not have to match. For example, the following code defines an interface that contains a method with a default parameter value of 3:

```
interface Igamma
{
  function doSomething(param:int = 3):void;
}
```

The following class definition implements the Igamma interface, but uses a different default parameter value:

```
class Gamma implements Igamma
{
  public function doSomething(param:int = 4):void {}
}
```

The reason for this flexibility is that the rules for implementing an interface are designed specifically to ensure data type compatibility, and requiring identical parameter names and default parameter values is not necessary to achieve that objective.

Inheritance

Inheritance is a form of code reuse that allows programmers to develop new classes that are based on existing classes. The existing classes are often referred to as *base classes* or *superclasses*, while the new classes are usually called *subclasses*. A key advantage of inheritance is that it allows you to reuse code from a base class yet leave the existing code unmodified. Moreover, inheritance requires no changes to the way that other classes interact with the base class. Rather than modifying an existing class that may have been thoroughly tested or may already be in use, using inheritance you can treat that class as an integrated module that you can extend with additional properties or methods. Accordingly, you use the `extends` keyword to indicate that a class inherits from another class.

Inheritance also allows you to take advantage of *polymorphism* in your code. Polymorphism is the ability to use a single method name for a method that behaves differently when applied to different data types. A simple example is a base class named Shape with two subclasses named Circle and Square. The Shape class defines a method named `area()`, which returns the area of the shape. If polymorphism is implemented, you can call the `area()` method on objects of type Circle and Square and have the correct calculations done for you. Inheritance enables polymorphism by allowing subclasses to inherit and redefine, or *override*, methods from the base class. In the following example, the `area()` method is redefined by the Circle and Square classes:

```
class Shape
{
  public function area():Number
  {
    return NaN;
  }
}

class Circle extends Shape
{
  private var radius:Number = 1;
  override public function area():Number
  {
    return (Math.PI * (radius * radius));
  }
}

class Square extends Shape
{
  private var side:Number = 1;
  override public function area():Number
  {
    return (side * side);
  }
}

var cir:Circle = new Circle();
trace(cir.area()); // 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // 1
```

Because each class defines a data type, the use of inheritance creates a special relationship between a base class and a class that extends it. A subclass is guaranteed to possess all the properties of its base class, which means that an instance of a subclass can always be substituted for an instance of the base class. For example, if a method defines a parameter of type Shape, it is legal to pass an argument of type Circle because Circle extends Shape, as in the following:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Instance properties and inheritance

An instance property, whether defined with the `function`, `var`, or `const` keywords, is inherited by all subclasses as long as the property is not declared with the `private` attribute in the base class. For example, the Event class in the Flash Player API has a number of subclasses that inherit properties common to all event objects.

For some types of events, the Event class contains all the properties necessary to define the event. These types of events do not require instance properties beyond those defined in the Event class. Examples of such events are the `complete` event, which occurs when data has loaded successfully, and the `connect` event, which occurs when a network connection has been established.

The following example is an excerpt from the Event class that shows some of the properties and methods that are inherited by subclasses. Because the properties are inherited, an instance of any subclass can access these properties.

```
public class Event
{
  public function get type():String;
  public function get bubbles():Boolean;
  ...

  public function stopPropagation():void {}
  public function stopImmediatePropagation():void {}
  public function preventDefault():void {}
  public function isDefaultPrevented():Boolean {}
```

```
    ...
}
```

Other types of events require unique properties not available in the Event class. These events are defined using subclasses of the Event class so that new properties can be added to the properties defined in the Event class. An example of such a subclass is the MouseEvent class, which adds properties unique to events associated with mouse movement or mouse clicks, such as the `mouseMove` and `click` events. The following example is an excerpt from the MouseEvent class that shows the definition of properties that exist on the subclass, but not on the base class:

```
public class MouseEvent extends Event
{
    public static const CLICK:String      = "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

Access control specifiers and inheritance

If a property is declared with the `public` keyword, the property is visible to code anywhere. This means that the `public` keyword, unlike the `private`, `protected`, and `internal` keywords, places no restrictions on property inheritance.

If a property is declared with `private` keyword, it is visible only in the class that defines it, which means that it is not inherited by any subclasses. This behavior is different from previous versions of ActionScript, where the `private` keyword behaved more like the ActionScript 3.0 `protected` keyword.

The `protected` keyword indicates that a property is visible not only within the class that defines it, but also to all subclasses. Unlike the `protected` keyword in the Java programming language, the `protected` keyword in ActionScript 3.0 does not make a property visible to all other classes in the same package. In ActionScript 3.0, only subclasses can access a property declared with the `protected` keyword. Moreover, a protected property is visible to a subclass whether the subclass is in the same package as the base class or in a different package.

To limit the visibility of a property to the package in which it is defined, use the `internal` keyword or do not use any access control specifier. The `internal` access control specifier is the default access control specifier that applies when one is not specified. A property marked as `internal` will be inherited only by a subclass that resides in the same package.

You can use the following example to see how each of the access control specifiers affects inheritance across package boundaries. The following code defines a main application class named AccessControl and two other classes named Base and Extender. The Base class is in a package named foo and the Extender class, which is a subclass of the Base class, is in a package named bar. The AccessControl class imports only the Extender class and creates an instance of Extender that attempts to access a variable named `str` that is defined in the Base class. The `str` variable is declared as `public` so that the code compiles and runs as shown in the following excerpt:

```
// Base.as in a folder named foo
package foo
{
  public class Base
  {
    public var str:String = "hello"; // change public on this line
  }
}

// Extender.as in a folder named bar
package bar
{
  import foo.Base;
  public class Extender extends Base
  {
    public function getString():String {
      return str;
    }
  }
}

// main application class in file named ProtectedExample.as
  import flash.display.MovieClip;
  import bar.Extender;
  public class AccessControl extends MovieClip
  {
    public function AccessControl()
    {
      var myExt:Extender = new Extender();
      trace (myExt.testString);  // error if str is not public
      trace (myExt.getString()); // error if str is private or internal
    }
  }
}
```

To see how the other access control specifiers affect compilation and execution of the preceding example, change the `str` variable's access control specifier to `private`, `protected`, or `internal` after deleting or commenting out the following line from the `AccessControl` class:

```
trace (myExt.testString);  // error if str is not public
```

Overriding variables not permitted

Properties that are declared with the `var` or `const` keywords are inherited, but cannot be overridden. To override a property means to redefine the property in a subclass. The only type of property that can be overridden are methods—that is, properties declared with the `function` keyword. Although you cannot override an instance variable, you can achieve similar functionality by creating getter and setter methods for the instance variable and overriding the methods. For more information, see "Overriding getters and setters" on page 135.

Overriding methods

To override a method means to redefine the behavior of an inherited method. Static methods are not inherited and cannot be overridden. Instance methods, however, are inherited by subclasses and can be overridden as long as the following two criteria are met:

- The instance method is not declared with the `final` keyword in the base class. When used with an instance method, the `final` keyword indicates the programmer's intent to prevent subclasses from overriding the method.
- The instance method is not declared with the `private` access control specifier in the base class. If a method is marked as `private` in the base class, there is no need to use the `override` keyword when defining an identically named method in the subclass because the base class method will not be visible to the subclass.

To override an instance method that meets these criteria, the method definition in the subclass must use the `override` keyword and must match the superclass version of the method in the following ways:

- The override method must have the same level of access control as the base class method. Methods marked as internal have the same level of access control as methods that have no access control specifier.
- The override method must have the same number of parameters as the base class method.
- The override method parameters must have the same data type annotations as the parameters in the base class method.
- The override method must have the same return type as the base class method.

The names of the parameters in the override method, however, do not have to match the names of the parameters in the base class, as long as the number of parameters and the data type of each parameter matches.

The super statement

When overriding a method, programmers often want to add to the behavior of the superclass method they are overriding instead of completely replacing the behavior. This requires a mechanism that allows a method in a subclass to call the superclass version of itself. The `super` statement provides such a mechanism, in that it contains a reference to the immediate superclass. The following example defines a class named Base that contains a method named `thanks()` and a subclass of the Base class named Extender that overrides the `thanks()` method. The `Extender.thanks()` method uses the `super` statement to call `Base.thanks()`.

```
package {
  import flash.display.MovieClip;
  public class SuperExample extends MovieClip
  {
    public function SuperExample()
    {
      var myExt:Extender = new Extender()
      trace(myExt.thanks()); // output: Mahalo nui loa
    }
  }
}

class Base {
  public function thanks():String
  {
    return "Mahalo";
  }
}

class Extender extends Base
{
  override public function thanks():String
  {
    return super.thanks() + " nui loa";
  }
}
```

Overriding getters and setters

Although you cannot override variables defined in a superclass, you can override getters and setters. For example, the following code overrides a getter named `currentLabel` that is defined in the MovieClip class in the Flash Player API.

```
package
{
  import flash.display.MovieClip;
  public class OverrideExample extends MovieClip
  {
    public function OverrideExample()
    {
      trace(currentLabel)
    }
    override public function get currentLabel():String
    {
      var str:String = "Override: ";
      str += super.currentLabel;
      return str;
    }
  }
}
```

The output of the `trace()` statement in the OverrideExample class constructor is `Override: null`, which shows that the example was able to override the inherited `currentLabel` property.

Static properties not inherited

Static properties are not inherited by subclasses. This means that static properties cannot be accessed through an instance of a subclass. A static property can be accessed only through the class object on which it is defined. For example, the following code defines a base class named Base and a subclass that extends Base named Extender. A static variable named `test` is defined in the Base class. The code as written in the following excerpt does not compile in strict mode and generates a run-time error in standard mode.

```
package {
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
      trace(myExt.test);  // error
    }
  }
}

class Base {
  public static var test:String = "static";
}

class Extender extends Base { }
```

The only way to access the static variable `test` is through the class object, as shown in the following code:

```
Base.test;
```

It is permissible, however, to define an instance property using the same name as a static property. Such an instance property can be defined in the same class as the static property or in a subclass. For example, the Base class in the preceding example could have an instance property named `test`. The following code compiles and executes because the instance property is inherited by the Extender class. The code would also compile and execute if the definition of the test instance variable is moved, but not copied, to the Extender class.

```
package
{
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
      trace(myExt.test);  // output: instance
    }
  }
}

class Base
{
  public static var test:String = "static";
  public var test:String = "instance";
}

class Extender extends Base {}
```

Static properties and the scope chain

Although static properties are not inherited, they are within the scope chain of the class that defines them and any subclass of that class. As such, static properties are said to be *in scope* of both the class in which they are defined and any subclasses. This means that a static property is directly accessible within the body of the class that defines the static property and any subclass of that class.

The following example modifies the classes defined in the previous example to show that the static `test` variable defined in the Base class is in scope of the Extender class. In other words, the Extender class can access the static `test` variable without prefixing the variable with the name of the class that defines `test`.

```
package
{
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
    }
  }
}

class Base {
  public static var test:String = "static";
}

class Extender extends Base
{
  public function Extender()
  {
    trace(test); // output: static
  }

}
```

If an instance property is defined that uses the same name as a static property in the same class or a superclass, the instance property has higher precedence in the scope chain. The instance property is said to *shadow* the static property, which means that the value of the instance property is used instead of the value of the static property. For example, the following code shows that if the Extender class defines an instance variable named test, the `trace()` statement uses the value of the instance variable instead of the value of the static variable.

```
package
{
  import flash.display.MovieClip;
  public class StaticExample extends MovieClip
  {
    public function StaticExample()
    {
      var myExt:Extender = new Extender();
    }
  }
}

class Base
{
  public static var test:String = "static";
}

class Extender extends Base
{
  public var test:String = "instance";
```

```
    public function Extender()
    {
        trace(test); // output: instance
    }

}
```

Advanced topics

This section begins with a brief history of ActionScript and OOP and continues with a discussion of the ActionScript 3.0 object model and how it enables the new ActionScript Virtual Machine (AVM2) to perform significantly faster than previous versions of Flash Player that contain the old ActionScript Virtual Machine (AVM1).

History of ActionScript OOP support

Because ActionScript 3.0 builds upon previous versions of ActionScript, it may be helpful to understand how the ActionScript object model has evolved. ActionScript began as a simple scripting mechanism for early versions of the Flash authoring tool. Subsequently, programmers began building increasingly complex applications with ActionScript. In response to the needs of such programmers, each subsequent release has added language features that facilitate the creation of complex applications.

ActionScript 1.0

ActionScript 1.0 refers to the version of the language used in Flash Player 6 and earlier. Even at this early stage of development, the ActionScript object model was based on the concept of the object as a fundamental data type. An ActionScript object is a compound data type with a group of *properties*. When discussing the object model, the term *properties* includes everything that is attached to an object, such as variables, functions or methods.

Although this first generation of ActionScript does not support the definition of classes with a `class` keyword, you can define a class using a special kind of object called a prototype object. Instead of using a `class` keyword to create an abstract class definition that you instantiate into concrete objects, as you do in class-based languages like Java and C++, prototype-based languages like ActionScript 1.0 use an existing object as a model (or prototype) for other objects. While objects in a class-based language may point to a class that serves as its template, objects in a prototype-based language point instead to another object, its prototype, that serves as its template.

To create a class in ActionScript 1.0, you define a constructor function for that class. In ActionScript, functions are actual objects, not just abstract definitions. The constructor function that you create serves as the prototypical object for instances of that class. The following code creates a class named Shape and defines one property named `visible` that is set to `true` by default:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

This constructor function defines a Shape class that you can instantiate with the `new` operator, as follows:

```
myShape = new Shape();
```

Just as the Shape constructor function object serves as the prototype for instances of the Shape class, it can also serve as the prototype for subclasses of Shape—that is, other classes that extend the Shape class.

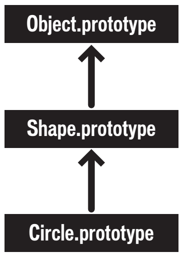
The creation of a class that is a subclass of the Shape class is a two-step process. First, create the class by defining a constructor function for the class, as follows:

```
// child class
function Circle(id, radius)
{
    this.id = id;
    this.radius = radius;
}
```

Second, use the `new` operator to declare that the Shape class is the prototype for the Circle class. By default, any class you create uses the Object class as its prototype, which means that `Circle.prototype` currently contains a generic object (an instance of the Object class). To specify that Circle's prototype is Shape instead of Object, use the following code to change the value of `Circle.prototype` so that it contains a Shape object instead of a generic object:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

The Shape class and the Circle class are now linked together in an inheritance relationship that is commonly known as the *prototype chain*. The diagram illustrates the relationships in a prototype chain:



The base class at the end of every prototype chain is the Object class. The Object class contains a static property named `Object.prototype` that points to the base prototype object for all objects created in ActionScript 1.0. The next object in our example prototype chain is the Shape object. This is because the `Shape.prototype` property was never explicitly set, so it still holds a generic object (an instance of the Object class). The final link in this chain is the Circle class, which is linked to its prototype, the Shape class (the `Circle.prototype` property holds a Shape object).

If we create an instance of the Circle class, as in the following example, the instance inherits the prototype chain of the Circle class:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Recall that we created a property named `visible` as a member of the Shape class. In our example, the `visible` property does not exist as a part of the `myCircle` object, only as a member of the Shape object, yet the following line of code outputs `true`:

```
trace(myCircle.visible); // true
```

Flash Player is able to ascertain that the `myCircle` object inherits the `visible` property by walking up the prototype chain. When executing this code, Flash Player first searches through the properties of the `myCircle` object for a property named `visible`, but does not find such a property. Flash Player looks next in the `Circle.prototype` object, but still does not find a property named `visible`. Continuing up the prototype chain, Flash Player finally finds the `visible` property defined on the `Shape.prototype` object and outputs the value of that property.

In the interest of simplicity, this section omits many of the details and intricacies of the prototype chain, and aims instead to provide enough information to help you understand the ActionScript 3.0 object model.

ActionScript 2.0

ActionScript 2.0 introduced new keywords such as `class`, `extends`, `public`, and `private`, that allowed you to define classes in a way that is familiar to anyone who works with class-based languages like Java and C++. It's important to understand that the underlying inheritance mechanism did not change between ActionScript 1.0 and ActionScript 2.0. ActionScript 2.0 merely added a new syntax for defining classes. The prototype chain works the same way in both versions of the language.

The new syntax introduced by ActionScript 2.0, shown in the following excerpt, allows you to define classes in a way that many programmers find more intuitive:

```
// base class
class Shape
{
    var visible:Boolean = true;
}
```

Note that ActionScript 2.0 also introduced type annotations for use with compile-time type checking. This allows you to declare that the `visible` property in the previous example should contain only a Boolean value. The new `extends` keyword also simplifies the process of creating a subclass. In the following example, the two-step process necessary in ActionScript 1.0 is accomplished in one step with the `extends` keyword:

```
// child class
class Circle extends Shape
{
  var id:Number;
  var radius:Number;
  function Circle(id, radius)
  {
    this.id = id;
    this.radius = radius;
  }
}
```

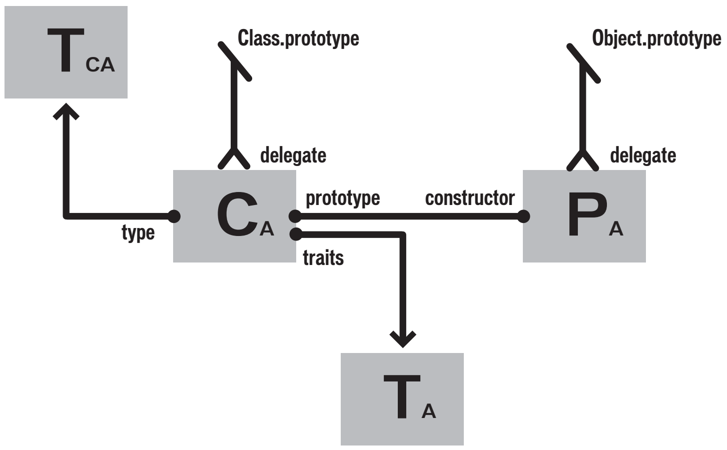
The constructor is now declared as part of the class definition, and the class properties `id` and `radius` must also be declared explicitly.

ActionScript 2.0 also added support for the definition of interfaces, which allow you to further refine your object-oriented programs with formally defined protocols for interobject communication.

The ActionScript 3.0 class object

A common object-oriented programming paradigm, most commonly associated with Java and C++, uses classes to define types of objects. Programming languages that adopt this paradigm also tend to use classes to construct instances of the data type that the class defines. ActionScript uses classes for both of these purposes, but its roots as a prototype-based language add an interesting characteristic. ActionScript creates for each class definition a special class object that allows sharing of both behavior and state. For many ActionScript programmers, however, this distinction may have no practical coding implications. ActionScript 3.0 is designed such that you can create sophisticated object-oriented ActionScript applications without using, or even understanding, these special class objects. For advanced programmers who want take advantage of class objects, this section discusses the issues in depth.

The following diagram shows the structure of a class object that represents a simple class named A that is defined with the statement `class A {}`:



Each rectangle in the diagram represents an object. Each object in the diagram has a subscript character A to represent that it belongs to class A. The class object (C_A) contains references to a number of other important objects. An instance traits object (T_A) stores the instance properties that are defined within a class definition. A class traits object (T_{CA}) represents the internal type of the class and stores the static properties defined by the class (the subscript character C stands for “class”). The prototype object (P_A) always refers to the class object to which it was originally attached through the `constructor` property.

The traits object

The traits object, which is new in ActionScript 3.0, was implemented with performance in mind. In previous versions of ActionScript, name lookup could be a time-consuming process as Flash Player walked the prototype chain. In ActionScript 3.0, name lookup is much more efficient and less time consuming because inherited properties are copied down from superclasses into the traits object of subclasses.

The traits object is not directly accessible to programmer code, but its presence can be felt by the improvements in performance and memory usage. The traits object provides the AVM2 with detailed information about the layout and contents of a class. With such knowledge, the AVM2 is able to significantly reduce execution time because it can often generate direct machine instructions to access properties or call methods directly without a time-consuming name lookup.

Thanks to the traits object, an object's memory footprint can be significantly smaller than a similar object in previous versions of ActionScript. For example, if a class is sealed (that is, the class is not declared dynamic), an instance of the class does not need a hash table for dynamically added properties, and can hold little more than a pointer to the traits objects and some slots for the fixed properties defined in the class. As a result, an object that required 100 bytes of memory in ActionScript 2.0 could require as little as 20 bytes of memory in ActionScript 3.0.

NOTE

The traits object is an internal implementation detail and there is no guarantee that it will not change or even disappear in future versions of ActionScript.

The prototype object

Every ActionScript class object has a property named `prototype`, which is a reference to the class's prototype object. The prototype object is a legacy of ActionScript's roots as prototype-based language. For more information, see "ActionScript 1.0" on page 139.

The `prototype` property is read-only, which means that it cannot be modified to point to different objects. This differs from the class `prototype` property in previous versions of ActionScript, where the prototype could be reassigned so that it pointed to a different class. Although the `prototype` property is read-only, the prototype object that it references is not. In other words, new properties can be added to the prototype object. Properties added to the prototype object are shared among all instances of the class.

The prototype chain, which was the only inheritance mechanism in previous versions of ActionScript, serves only a secondary role in ActionScript 3.0. The primary inheritance mechanism, fixed property inheritance, is handled internally by the traits object. A fixed property is a variable or method that is defined as part of a class definition. Fixed property inheritance is also called class inheritance because it is the inheritance mechanism that is associated with keywords such as `class`, `extends`, and `override`.

The prototype chain provides an alternative inheritance mechanism that is more dynamic than fixed property inheritance. You can add properties to a class's prototype object not only as part of the class definition, but also at run time through the class object's `prototype` property. Note, however, that if you set the compiler to strict mode, you may not be able to access properties added to a prototype object unless you declare a class with the `dynamic` keyword.

A good example of a class with several properties attached to the prototype object is the Object class. The Object class's `toString()` and `valueOf()` methods are actually functions assigned to properties of the Object class's prototype object. The following is an example of how the declaration of these methods could, in theory, look (the actual implementation differs slightly because of implementation details):

```
public dynamic class Object
{
  prototype.toString = function()
  {
    // statements
  };
  prototype.valueOf = function()
  {
    // statements
  };
}
```

As mentioned previously, you can attach a property to a class's prototype object outside the class definition. For example, the `toString()` method can also be defined outside the Object class definition, as follows:

```
Object.prototype.toString = function()
{
  // statements
};
```

Unlike fixed property inheritance, however, prototype inheritance does not require the `override` keyword if you want to redefine a method in a subclass. For example. if you want to redefine the `valueOf()` method in a subclass of the Object class, you have three options. First, you can define a `valueOf()` method on the subclass's prototype object inside the class definition. The following code creates a subclass of Object named Foo and redefines the `valueOf()` method on Foo's prototype object as part of the class definition. Because every class inherits from Object, it is not necessary to use the `extends` keyword.

```
dynamic class Foo
{
  prototype.valueOf = function()
  {
    return "Instance of Foo";
  };
}
```

Second, you can define a `valueOf()` method on Foo's prototype object outside the class definition, as shown in the following code:

```
Foo.prototype.valueOf = function()
{
  return "Instance of Foo";
};
```

Third, you can define a fixed property named `valueOf()` as part of the Foo class. This technique differs from the others in that it mixes fixed property inheritance with prototype inheritance. Any subclass of Foo that wants to redefine `valueOf()` must use the `override` keyword. The following code shows `valueOf()` defined as a fixed property in Foo:

```
class Foo
{
  function valueOf() {
    return "Instance of Foo";
  }
}
```

The AS3 namespace

The existence of two separate inheritance mechanisms, fixed property inheritance and prototype inheritance, creates an interesting compatibility challenge with respect to the properties and methods of the core classes. Compatibility with the ECMAScript, Edition 4 draft language specification requires the use of prototype inheritance, which means that the properties and methods of a core class are defined on the prototype object of that class. On the other hand, compatibility with the Flash Player API calls for the use of fixed property inheritance, which means that the properties and methods of a core class are defined in the class definition using the `const`, `var`, and `function` keywords. Moreover, the use of fixed properties instead of the prototype versions can lead to significant increases in run-time performance.

ActionScript 3.0 solves this problem by using both prototype inheritance and fixed property inheritance for the core classes. Each core class contains two sets of properties and methods. One set is defined on the prototype object for compatibility with the ECMAScript specification, and the other set is defined with fixed properties and the AS3 namespace for compatibility with the Flash Player API.

The AS3 namespace provides a convenient mechanism for choosing between the two sets of properties and methods. If you do not use the AS3 namespace, an instance of a core class inherits the properties and methods defined on the core class's prototype object. If you decide to use the AS3 namespace, an instance of a core class inherits the AS3 versions because fixed properties are always preferred over prototype properties. In other words, whenever a fixed property is available, it is always used instead of an identically named prototype property.

You can selectively use the AS3 namespace version of a property or method by qualifying it with the AS3 namespace. For example, the following code uses the AS3 version of the `Array.pop()` method:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Alternatively, you can use the `use namespace` directive to open the AS3 namespace for all the definitions within a block of code. For example, the following code uses the `use namespace` directive to open the AS3 namespace for both the `pop()` and `push()` methods:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 also provides compiler options for each set of properties so that you can apply the AS3 namespace to your entire program. The `-as3` compiler option represents the AS3 namespace, and the `-es` compiler option represents the prototype inheritance option (`es` stands for ECMAScript). To open the AS3 namespace for your entire program, set the `-as3` compiler option to `true`, and the `-es` compiler option to `false`. To use the prototype versions, set the compiler options to the opposite values. The default compiler settings for Adobe Flex Builder 2 and Adobe Flash CS3 Professional are `-as3 = true` and `-es = false`.

If you plan to extend any of the core classes and override any methods, you should understand how the AS3 namespace can affect how you must declare an overridden method. If you are using the AS3 namespace, any method override of a core class method must also use the AS3 namespace, along with the `override` attribute. If you are not using the AS3 namespace and want to redefine a core class method in a subclass, you should not use the AS3 namespace or the `override` keyword.

Example: GeometricShapes

The GeometricShapes example application shows how a number of object-oriented concepts and features can be applied using ActionScript 3.0, including:

- Defining classes
- Extending classes
- Polymorphism and the `override` keyword
- Defining, extending and implementing interfaces

It also includes a "factory method" that creates class instances, showing how to declare a return value as an instance of an interface, and use that returned object in a generic way.

The GeometricShapes application files can be found in the folder Examples/ GeometricShapes. The application consists of the following files:

File	Description
GeometricShapes.mxml	The main application file in MXML for Flex.
com/example/programmingas3/ geometricshapes/IGeometricShape.as	The base interface defining methods to be implemented by all GeometricShapes application classes.
com/example/programmingas3/ geometricshapes/IPolygon.as	An interface defining methods to be implemented by GeometricShapes application classes that have multiple sides.

File	Description
com/example/programmingas3/ geometricshapes/RegularPolygon.as	A type of geometric shape that has sides of equal length postioned symmetrically around the shape's center.
com/example/programmingas3/ geometricshapes/Circle.as	A type of geometric shape that defines a circle.
com/example/programmingas3/ geometricshapes/EquilateralTriangle.as	A subclass of RegularPolygon that defines a triangle with all sides the same length.
com/example/programmingas3/ geometricshapes/Square.as	A subclass of RegularPolygon defining a rectangle with all four sides the same length.
com/example/programmingas3/ geometricshapes/ GeometricShapeFactory.as	A class containing a factory method for creating shapes given a shape type and size.

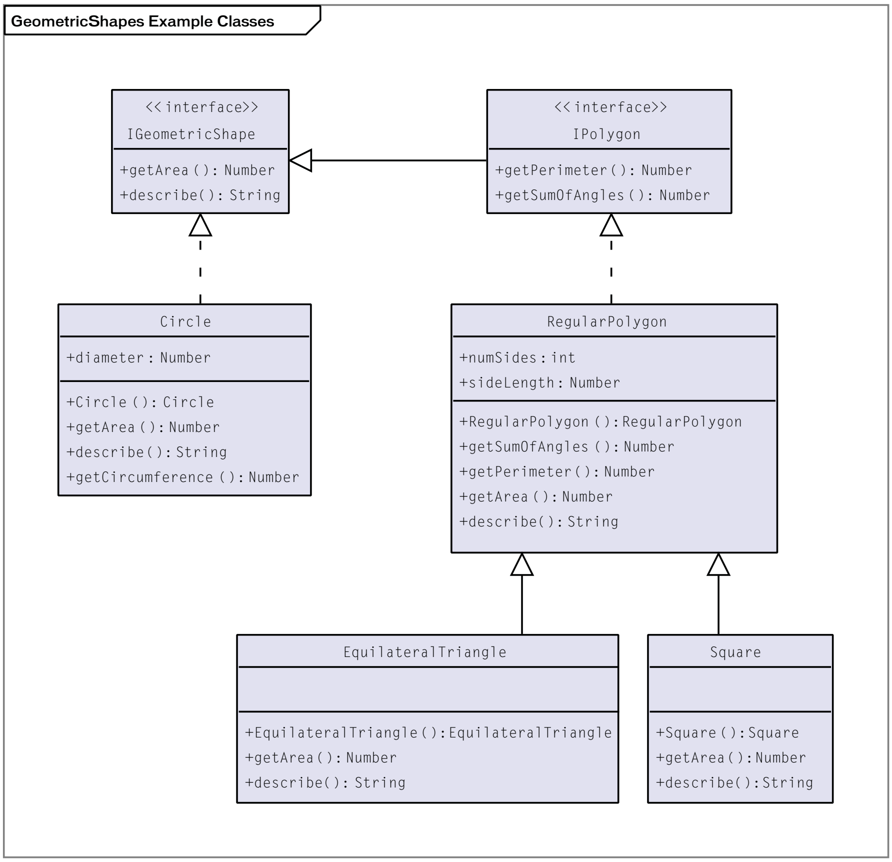
Defining the GeometricShapes classes

The GeometricShapes application lets the user specify a type of geometric shape and a size. It then responds with a description of the shape, its area, and distance around its perimeter.

The application user interface is trivial, including a few controls for selecting the type of shape, setting the size, and displaying the description. The most interesting part of this application is under the surface, in the structure of the classes and interfaces themselves.

This application deals with geometric shapes, but it doesn't display them graphically. It provides a small library of classes and interfaces that will be reused in a later chapter's "Example: SpriteArranger" on page 187. The Sprite Arranger example displays the shapes graphically and lets the user manipulate them, based on the class framework provided here in the GeometricShapes application.

The classes and interfaces that define the geometric shapes in this example are shown in the following diagram using Unified Modeling Language (UML) notation:



Defining common behavior with interfaces

This GeometricShapes application deals with three types of shapes: circles, squares, and equilateral triangles. The GeometricShapes class structure begins with a very simple interface, IGeometricShape, that lists methods common to all three types of shapes:

```
package com.example.programmingas3.geometricshapes
{
  public interface IGeometricShape
  {
    function getArea():Number;
    function describe():String;
    }
}
```

The interface defines two methods: the `getArea()` method, which calculates and returns the area of the shape, and the `describe()` method, which assembles a text description of the shape's properties.

We also want to know the distance around the perimeter of each shape. However, the perimeter of a circle is called the circumference and it's calculated in a unique way, so the behavior diverges from that of a triangle or a square. Still there is enough similarity between triangles, squares, and other polygons that it makes sense to define a new interface class just for them: IPolygon. The IPolygon interface is also rather simple, as shown here:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

This interface defines two methods common to all polygons: the `getPerimeter()` method that measures the combined distance of all the sides, and the `getSumOfAngles()` method that adds up all the interior angles.

The IPolygon interface extends the IGeometricShape interface, which means that any class that implements the IPolygon interface must declare all four methods—the two from the IGeometricShape interface, and the two from the IPolygon interface.

Defining the shape classes

Once you have a good idea about the methods common to each type of shape, you can define the shape classes themselves. In terms of how many methods you need to implement, the simplest shape is the Circle class, shown here:

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius^2.
            return Math.PI * ((diameter / 2)^2);
        }

        public function getCircumference():Number
        {
            // The formula is Pi * radius * 2.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

The Circle class implements the IGeometricShape intrface, so it must provide code for both the `getArea()` method and the `describe()` method. In addition, it defines the `getCircumference()` method, which is unique to the Circle class. The Circle class also declares a property, `diameter`, which won't be found in the other polygon classes.

The other two types of shapes, squares and equilateral triangles, have some other things in common: they each have sides of equal length, and there are common formulas you can use to calculate the perimeter and sum of interior angles for both. In fact, those common formulas will apply to any other regular polygons that you need to define in the future as well.

The RegularPolygon class will be the superclass for both the Square class and the EquilateralTriangle class. A superclass lets you define common methods in one place, so you don't have to define them separately in each subclass. Here is the code for the RegularPolygon class:

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }
        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
            {
              return ((numSides - 2) * 180);
            }
            else
            {
              return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " +
    getSumOfAngles() + " degrees.\n";
            return desc;
        }
    }
}
```

First, the RegularPolygon class declares two properties that are common to all regular polygons: the length of each side (the `sideLength` property) and the number of sides (the `numSides` property).

The RegularPolygon class implements the IPolygon interface and declares all four of the IPolygon interface methods. It implements two of these—the `getPerimeter()` and `getSumOfAngles()` methods—using common formulas.

Because the formula for the `getArea()` method will differ from shape to shape, the base class version of the method cannot include common logic that can be inherited by the subclass methods. Instead, it simply returns a 0 default value to indicate that the area was not calculated. To calculate the area of each shape correctly, the subclasses of the RegularPolygon class will have to override the `getArea()` method themselves.

The following code for the EquilateralTriangle class show how the `getArea()` method is overridden:

```
package com.example.programmingas3.geometricshapes
{
  public class EquilateralTriangle extends RegularPolygon
  {
    public function EquilateralTriangle(len:Number = 100):void
    {
      super(len, 3);
    }

    public override function getArea():Number
    {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
    }

    public override function describe():String
    {
         /* starts with the name of the shape, then delegates the rest
         of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
    }
  }
}
```

The `override` keyword indicates that the `EquilateralTriangle.getArea()` method intentionally overrides the `getArea()` method from the RegularPolygon superclass. When the `EquilateralTriangle.getArea()` method is called, it calculates the area using the formula in the preceding code, and the code in the `RegularPolygon.getArea()` method never executes.

In contrast, the EquilateralTriangle class doesn't define its own version of the `getPerimeter()` method. When the `EquilateralTriangle.getPerimeter()` method is called, the call goes up the inheritance chain and executes the code in the `getPerimeter()` method of the RegularPolygon superclass.

The `EquilateralTriangle()` contructor uses the `super()` statement to explicitly invoke the `RegularPolygon()` contructor of its superclass. If both constructors had the same set of parameters, you could have omitted the EquilateralTriangle constructor completely, and the `RegularPolygon()` contructor would be executed instead. However, the `RegularPolygon()` contructor needs an extra parameter, `numSides`. So the `EquilateralTriangle()` constructor calls `super(len, 3)` which passes along the `len` input parameter and the value 3 to indicate that the triangle will have 3 sides.

The `describe()` method also uses the `super()` statement, but in a different way—to invoke the RegularPolygon superclass' version of the `describe()` method. The `EquilateralTriangle.describe()` method first sets the `desc` string variable to a statement about the type of shape. Then it gets the results of the `RegularPolygon.describe()` method by calling `super.describe()`, and it appends that result to the `desc` string.

The Square class won't be described in detail here, but it is similar to the EquilateralTriangle class, providing a constructor and its own implementations of the `getArea()` and `describe()` methods.

Polymorphism and the factory method

A set of classes that make good use of interfaces and inheritance can be used in many interesting ways. For example, all of the shape classes described so far either implement the IGeometricShape interface or extend a superclass that does. So if you define a variable to be an instance of IGeometricShape, you don't have to know whether it is actually an instance of the Circle or the Square class in order to call its `describe()` method.

The following code shows how this works:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

When `myShape.describe()` is called, it executes the method `Circle.describe()` because even though the variable is defined as an instance of the IGeometricShape interface, Circle is its underlying class.

This example shows the principle of polymorphism in action: the exact same method call results in different code being executed, depending on the class of the object whose method is being invoked.

The GeometricShapes application applies this kind of interface-based polymorphism using a simplified version of a design pattern known as the factory method. The term *factory method* refers to a function that returns an object whose underlying data type or contents can differ depending on the context.

The GeometricShapeFactory class shown here defines a factory method named `createShape()`:

```
package com.example.programmingas3.geometricshapes
{
  public class GeometricShapeFactory
  {
    public static var currentShape:IGeometricShape;

    public static function createShape(shapeName:String,
                                       len:Number):IGeometricShape
    {
      switch (shapeName)
      {
        case "Triangle":
          return new EquilateralTriangle(len);

        case "Square":
          return new Square(len);

        case "Circle":
          return new Circle(len);
      }
      return null;
    }

    public static function describeShape(shapeType:String,
  shapeSize:Number):String
    {
      GeometricShapeFactory.currentShape =
        GeometricShapeFactory.createShape(shapeType, shapeSize);
      return GeometricShapeFactory.currentShape.describe();
    }
  }
}
```

The `createShape()` factory method lets the shape subclass constructors define the details of the instances that they create, while returning the new objects as IGeometricShape instances so that they can be handled by the application in a more general way.

The `describeShape()` method in the preceding example shows how an application can use the factory method to get a generic reference to a more specific object. The application can get the description for a newly created Circle object like this:

```
GeometricShapeFactory.describeShape(“Circle”, 100);
```

The `describeShape()` method then calls the `createShape()` factory method with the same parameters, storing the new Circle object in a static variable named `currentShape`, which was typed as an IGeometricShape object. Next, the `describe()` method is called on the `currentShape` object, and that call is automatically resolved to execute the `Circle.describe()` method, returning a detailed description of the circle.

Enhancing the example application

The real power of interfaces and inheritance becomes apparent when you enhance or change your application.

Say that you wanted to add a new shape, a pentagon, to this example application. You would create a new Pentagon class that extends the RegularPolygon class and defines its own versions of the `getArea()` and `describe()` methods. Then you would add a new Pentagon option to the combo box in the application's user interface. But that's it. The Pentagon class would automatically get the functionality of the `getPerimeter()` method and the `getSumOfAngles()` method from the RegularPolygon class by inheritance. Because it inherits from a class that implements the IGeometricShape interface, a Pentagon instance can be treated as an IGeometricShape instance too. That means you do not need to change any of the methods in the GeometricShapeFactory class, making it much easier to add new types of shapes when needed.

You may want to add a Pentagon class to the Geometric Shapes example as an exercise, to see how interfaces and inheritance can ease the workload of adding new features to an application.

CHAPTER 5

Display Programming

5

Display programming in ActionScript 3.0 allows you to work with elements that appear on the Stage of Adobe Flash Player 9. This chapter describes the basic concepts for working with onscreen elements. You'll learn the details about programmatically organizing visual elements. You'll also learn about creating your own custom classes for display objects.

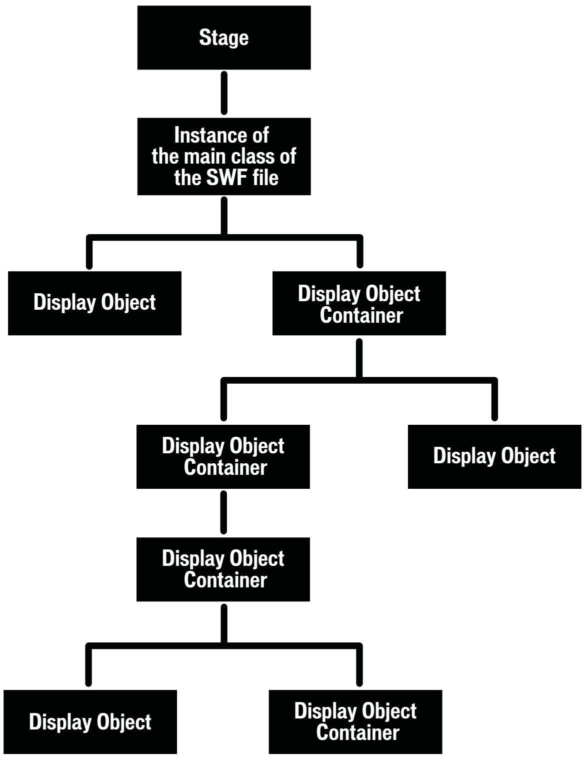
Contents

Understanding the display architecture

Each application built with ActionScript 3.0 has a hierarchy of displayed objects known as the *display list*. The display list contains all the visible elements in the application. Display elements fall into one or more of the following groups:

- The Stage

  The Stage is the base container of display objects. Each application has one Stage object, which contains all onscreen display objects. The Stage is the top-level container and is at the top of the display list hierarchy:



  Each SWF file has an associated ActionScript class, known as *the main class of the SWF file*. When you embed a SWF file in an HTML page, Flash Player calls the constructor function for that class and the instance that is created (which is always a type of display object) is added as a child of the Stage object. The main class of a SWF file always extends the Sprite class (for more information, see "Core display classes" on page 162).

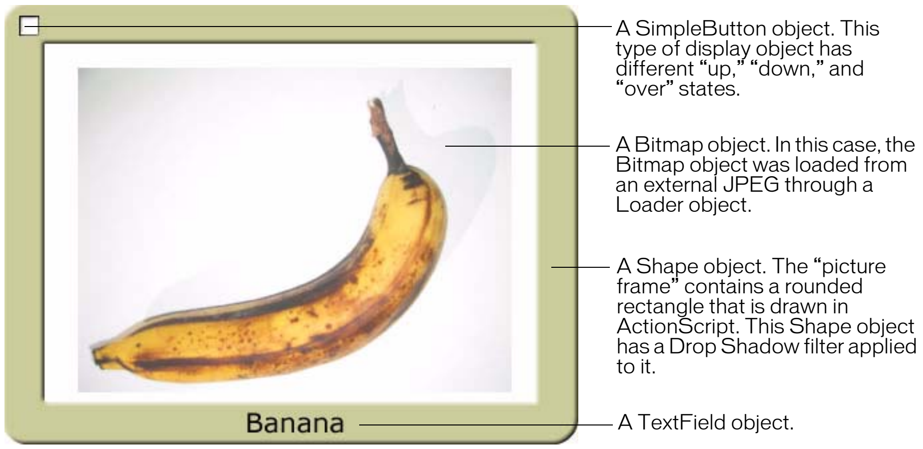
You can access the Stage through the `stage` property of any DisplayObject instance. For more information, see "Setting Stage properties" on page 173.

- Display objects

  In ActionScript 3.0, all elements that appear on screen in an application are types of *display objects*. The flash.display package includes a DisplayObject class, which is a base class extended by a number of other classes. These different classes represent different types of display objects, such as vector shapes, movie clips, and text fields, to name a few. For an overview of these classes, see "Core display classes" on page 162.
- Display object containers

  Display object containers are special types of display objects that can contain child objects that are also display objects.

  The DisplayObjectContainer class is a subclass of the DisplayObject class. A DisplayObjectContainer object can contain multiple display objects in its *child list*. For example, the following illustration shows a type of DisplayObjectContainer object known as a Sprite that contains various display objects:

  
  A SimpleButton object. This type of display object has different "up," "down," and "over" states.
A Bitmap object. In this case, the Bitmap object was loaded from an external JPEG through a Loader object.
A Shape object. The "picture frame" contains a rounded rectangle that is drawn in ActionScript. This Shape object has a Drop Shadow filter applied to it.
Banana
A TextField object.

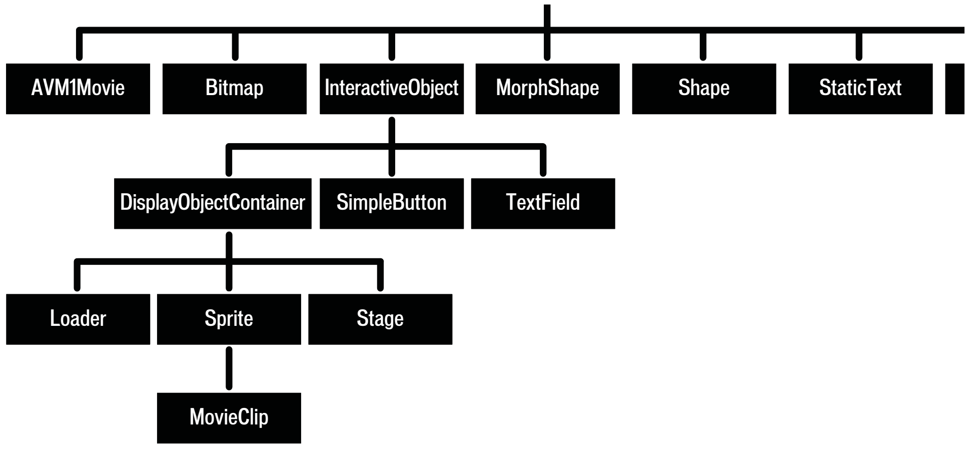
  In the context of discussing display objects, DisplayObjectContainer objects are also known as *display object containers* or simply *containers*.

  Although all visible display objects inherit from the DisplayObject class, the type of each is of a specific subclass of DisplayObject class. For example, there is a constructor function for the Shape class or the Video class, but there is no constructor function for the DisplayObject class.

  As noted earlier, the Stage is a display object container.

Core display classes

The ActionScript 3.0 flash.display package includes classes for visual objects that can appear in Flash Player. The following illustration shows the subclass relationships of these core display object classes.



The illustration shows the class inheritance of display object classes. Note that some of these classes, specifically StaticText, TextField, and Video, are not in the flash.display package, but they still inherit from the DisplayObject class.

All classes that extend the DisplayObject class inherit its methods and properties. For more information, see "Properties and methods of the DisplayObject class" on page 167.

You can instantiate objects of the following classes contained in the flash.display package:

- Bitmap—You use the Bitmap class to define bitmap objects, either loaded from external files or rendered through ActionScript. You can load bitmaps from external files through the Loader class. You can load GIF, JPG, or PNG files. You can also create a BitmapData object with custom data and then create a Bitmap object that uses that data. You can use the methods of the BitmapData class to alter bitmaps, whether they are loaded or created in ActionScript. For more information, see "The Loader class" on page 179 and "Creating and manipulating bitmaps" on page 184.
- Loader—You use the Loader class to load external assets (either SWF files or graphics). For more information, see "Loading content dynamically" on page 179.
- Shape—You use the Shape class to create vector graphics, such as rectangles, lines, circles, and so on. For more information, see "Drawing vector graphics" on page 176.
- SimpleButton—A SimpleButton object has three button states: up, down, and over. For more information, see "Working with SimpleButton objects" on page 184.

- Sprite—A Sprite object can contain graphics of its own, and it can contain child display objects. (The Sprite class extends the DisplayObjectContainer class). For more information, see "Working with display object containers" on page 168 and "Drawing vector graphics" on page 176.
- MovieClip—A MovieClip object is similar to a Sprite object, except that it also has a timeline. For more information, see "Controlling ActionScript 3.0 movie clips" on page 183.

The following classes, which are not in the flash.display package, are subclasses of the DisplayObject class:

- The TextField class, included in the flash.text package—TextField objects are display objects for text display and input. For more information, see "Working with text" on page 178.
- The Video class, included in the flash.media package—see "Working with video" on page 186.

You cannot instantiate instances of the following classes in the flash.display package, but they do extend the DisplayObject class:

- AVM1Movie—The AVM1Movie class is used to represent loaded SWF files that are authored in ActionScript 1.0 and 2.0.
- DisplayObjectContainer—The Loader, Stage, Sprite, and MovieClip classes each extend the DisplayObjectContainer class. For more information, see "Working with display object containers" on page 168.
- InteractiveObject—InteractiveObject is the base class for all objects used to interact with the mouse and keyboard. SimpleButton, TextField, Video, Loader, Sprite, Stage, and MovieClip objects are all subclasses of the InteractiveObject class.
- MorphShape—These objects are created when you apply a shape tween. You cannot instantiate them using ActionScript. Morph shapes are created only in the Flash authoring tool.
- Stage—The Stage class extends the DisplayObjectContainer class. There is one Stage instance for an application, and it is at the top of the display list hierarchy. For more information, see "Setting Stage properties" on page 173.

Also, the StaticText class, in the flash.text package, extends the DisplayObject class, but you cannot instantiate an instance of it. Static text fields are created only in the Flash authoring tool.

Advantages of the ActionScript 3.0 display list approach

In ActionScript 3.0, there are separate classes for different types of display objects. In ActionScript 1.0 and 2.0, many of the same types of objects are all included in one class: the MovieClip class.

This individualization of classes and the hierarchical structure of display lists have the following benefits:

- More efficient rendering and reduced memory usage
- Improved depth management
- Full traversal of the display list
- Off-list display objects
- Easier subclassing of display objects

These benefits are described in the next sections.

More efficient rendering and smaller file sizes

In ActionScript 1.0 and 2.0, you could draw shapes only in a MovieClip object. In ActionScript 3.0, there are simpler display object classes in which you can draw shapes. Because these ActionScript 3.0 display object classes do not include the full set of methods and properties that a MovieClip object includes, they are less taxing on memory and processor resources.

For example, each MovieClip object includes properties for the timeline of the movie clip, whereas a Shape object does not. The properties for managing the timeline can use a lot of memory and processor resources. In ActionScript 3.0, using the Shape object results in better performance. The Shape object has less overhead than the more complex MovieClip object. Flash Player does not need to manage unused MovieClip properties, which improves speed and reduces the memory footprint the object uses.

Improved depth management

In ActionScript 1.0 and 2.0, depth was managed through a linear depth management scheme and methods such as `getNextHighestDepth()`.

ActionScript 3.0 includes the DisplayObjectContainer class, which has more convenient methods and properties for managing the depth of display objects.

In ActionScript 3.0, when you move a display object to a new position in the child list of a DisplayObjectContainer instance, the other children in the display object container are repositioned automatically and assigned appropriate child index positions in the display object container.

Also, in ActionScript 3.0 it is always possible to discover all of the child objects of any display object container. Every DisplayObjectContainer instance has a `numChildren` property, which lists the number of children in the display object container. And since the child list of a display object container is always an indexed list, you can examine every object in the list from index position 0 through the last index position (`numChildren - 1`). This was not possible with the methods and properties of a MovieClip object in ActionScript 1.0 and 2.0.

In ActionScript 3.0, you can easily traverse the display list sequentially; there are no gaps in the index numbers of a child list of a display object container. Traversing the display list and managing the depth of objects is much easier than was possible in ActionScript 1.0 and 2.0. In ActionScript 1.0 and 2.0, a movie clip could contain objects with intermittent gaps in the depth order, which could make it difficult to traverse the list of object. In ActionScript 3.0, each child list of a display object container is cached internally as an array, resulting in very fast lookups (by index). Looping through all children of a display object container is also very fast.

In ActionScript 3.0, you can also access children in a display object container by using the `getChildByName()` method of the DisplayObjectContainer class.

Full traversal of the display list

In ActionScript 1.0 and 2.0, you could not access some objects, such as vector shapes, that were drawn in the Flash authoring tool. In ActionScript 3.0, you can access all objects on the display list—both those created using ActionScript and all display objects created in the Flash authoring tool. For details, see “Traversing the display list” on page 171.

Off-list display objects

In ActionScript 3.0, you can create display objects that are not on the visible display list. These are known as *off-list* display objects. A display object is added to the visible display list only when you call the `addChild()` or `addChildAt()` method of a DisplayObjectContainer instance that has already been added to the display list.

You can use off-list display objects to assemble complex display objects, such as those that have multiple display object containers containing multiple display objects. By keeping display objects off-list, you can assemble complicated objects without using the processing time to render these display objects. You can then add an off-list display object to the display list when it is needed. Also, you can move a child of a display object container on and off the display list and to any desired position in the display list at will.

Easier subclassing of display objects

In ActionScript 1.0 and 2.0, you would often have to add new MovieClip objects to a SWF file to create basic shapes or to display bitmaps. In ActionScript 3.0, the DisplayObject class includes many built-in subclasses, including Shape and Bitmap. Because the classes in ActionScript 3.0 are more specialized for specific types of objects, it is easier to create basic subclasses of the built-in classes.

For example, in order to draw a circle in ActionScript 2.0, you could create a CustomCircle class that extends the MovieClip class when an object of the custom class is instantiated. However, that class would also include a number of properties and methods from the MovieClip class (such as `totalFrames`) that do not apply to the class. In ActionScript 3.0, however, you can create a CustomCircle class that extends the Shape object, and as such does not include the unrelated properties and methods that are contained in the MovieClip class. The following code shows an example of a CustomCircle class:

```
import flash.display.*;

private class CustomCircle extends Shape
{
  var xPos:Number;
  var yPos:Number;
  var radius:Number;
  var color:uint;
  public function CustomCircle(xInput:Number,
                      yInput:Number,
                      rInput:Number,
                      colorInput:uint)
  {
    xPos = xInput;
    yPos = yInput;
    radius = rInput;
    color = colorInput;
    this.graphics.beginFill(color);
    this.graphics.drawCircle(xPos, yPos, radius);
  }
}
```

Working with display objects

Now that you understand the basic concepts of the Stage, display objects, display object containers, and the display list, this section provides you with some more specific information about working with display objects in ActionScript 3.0.

Properties and methods of the DisplayObject class

All display objects are subclasses of the DisplayObject class, and as such they inherit the properties and methods of the DisplayObject class. The properties inherited are basic properties that apply to all display objects. For example, each display object has an `x` property and a `y` property that specifies the object's position in its display object container.

There is no constructor function for the DisplayObject class. You must create another type of object (an object that is a subclass of the DisplayObject type), such as a Sprite, to instantiate an object with the `new` constructor. Also, if you want to create a custom display object class, you must create a subclass of one of the display object subclasses that has a usable constructor function (such as the Shape class or the Sprite class).

Adding display objects to the display list

When you instantiate a display object, it will not appear onscreen (on the Stage) until you add the display object instance to a display object container that is on the display list. For example, in the following code, the `myText` TextField object would not be visible if you omitted the last line of code. In the last line of code, the `this` keyword must refer to a display object container that is already added to the display list.

```
import flash.display.*;
import flash.text.TextField;
var myText:TextField = new TextField();
myText.text = "Buenos dias.";
this.addChild(myText);
```

When you add any visual element to the Stage, that element becomes a *child* of the Stage object. The first SWF file loaded in an application (for example, the one that you embed in an HTML page) is automatically added as a child of the Stage. It can be an object of any type that extends the Sprite class.

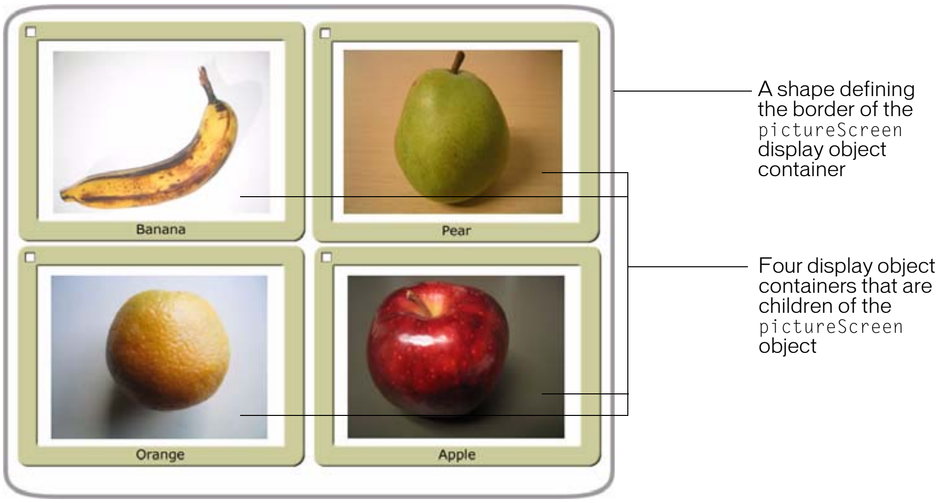
Any display objects that you create *without* using ActionScript—for example, by adding an MXML tag in Adobe Flex Builder 2 or by using a drawing tool in Flash—are added to the display list. Although you do not add these display objects through ActionScript, you can access them through ActionScript. For example, the following code adjusts the width of an object named `button1` that was added in the authoring tool (not through ActionScript):

```
button1.width = 200;
```

Working with display object containers

If a DisplayObjectContainer object is deleted from the display list, or if it is moved or transformed in some other way, each display object in the DisplayObjectContainer is also deleted, moved, or transformed.

A display object container is itself a type of display object—it can be added to another display object container. For example, the following image shows a display object container, `pictureScreen`, that contains one outline shape and four other display object containers (of type PictureFrame):


Banana
Pear
Orange
Apple
A shape defining the border of the pictureScreen display object container
Four display object containers that are children of the pictureScreen object

In order to have a display object appear in the display list, you must add it to a display object container that is on the display list. You do this by using the `addChild()` method or the `addChildAt()` method of the container object. For example, without the final line of the following code, the `myTextField` object would not be displayed:

```
var myTextField:TextField = new TextField();
```

```
myTextField.text = "hello";
this.root.addChild(myTextField);
```

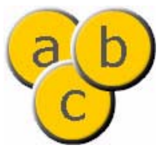
In this code sample, `this.root` points to the MovieClip display object container that contains the code. In your actual code, you may specify a different container.

Use the `addChildAt()` method to add the child to a specific position in the child list of the display object container. These zero-based index positions in the child list relate to the layering (the front-to-back order) of the display objects. For example, consider the following three display objects. Each object was created from a custom class called Ball.

The layering of these display objects in their container can be adjusted using the `addChildAt()` method. For example, consider the following code:

```
ball_A = new Ball(0xFFCC00, "a");
ball_A.name = "ball_A";
ball_A.x = 20;
ball_A.y = 20;
container.addChild(ball_A);

ball_B = new Ball(0xFFCC00, "b");
ball_B.name = "ball_B";
ball_B.x = 70;
ball_B.y = 20;
container.addChild(ball_B);

ball_C = new Ball(0xFFCC00, "c");
ball_C.name = "ball_C";
ball_C.x = 40;
ball_C.y = 60;
container.addChildAt(ball_C, 1);
```

After executing this code, the display objects are positioned as follows in the `container` DisplayObjectContainer object. Notice the layering of the objects.



You can use the `getChildAt()` method to verify the layer order of the display objects. The `getChildAt()` method returns child objects of a container based on the index number you pass it. For example, the following code reveals names of display objects at different positions in the child list of the `container` DisplayObjectContainer object:

```
trace(container.getChildAt(0).name); // ball_A
trace(container.getChildAt(1).name); // ball_C
trace(container.getChildAt(2).name); // ball_B
```

If you remove a display object from the parent container's child list, the higher elements on the list each move down a position in the child index. For example, continuing with the previous code, the following code shows how the display object that was at position 2 in the `container` DisplayObjectContainer moves to position 1 if a display object that is lower in the child list is removed:

```
container.removeChild(ball_C);
trace(container.getChildAt(0).name); // ball_A
trace(container.getChildAt(1).name); // ball_B
```

The `removeChild()` and `removeChildAt()` methods do not delete a display object instance entirely. They simply remove it from the child list of the container. The instance can still be referenced by another variable. (Use the `delete` operator to completely remove an object.)

Because a display object has only one parent container, you can add an instance of a display object to only one display object container. For example, the following code shows that the display object `tf1` can exist in only one container (in this case, a Sprite, which extends the DisplayObjectContainer class):

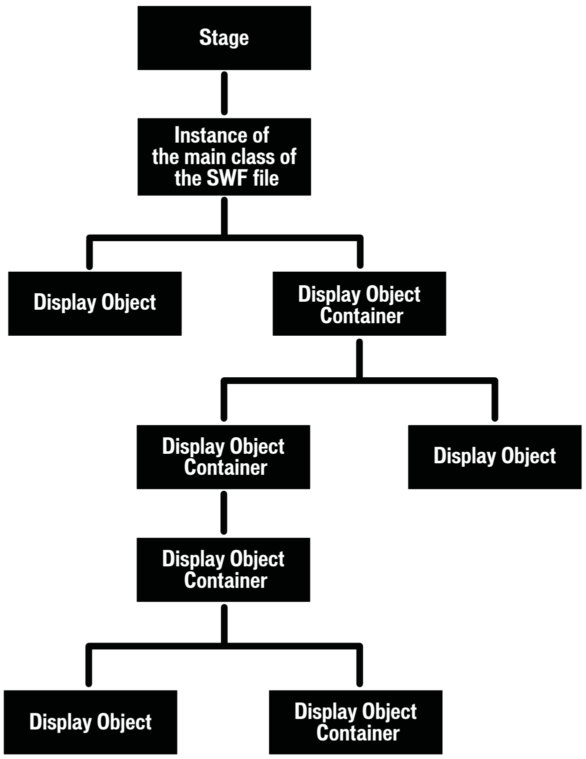
```
tf1:TextField = new TextField();
tf2:TextField = new TextField();
tf1.name = "text 1";
tf2.name = "text 2";

container1:Sprite = new Sprite();
container2:Sprite = new Sprite();

container1.addChild(tf1);
container1.addChild(tf2);
container2.addChild(tf1);

trace(container1.numChildren); // 1
trace(container1.getChildAt(0).name); // text 2
trace(container2.numChildren); // 1
trace(container2.getChildAt(0).name); // text 1
```

If you add a display object that is contained in one display object container to another display object container, it is removed from the first display object container's child list.

Recall that a display object that is off the display list—one that is not included in a display object container that is a child of the Stage—is known as an *off-list* display object.

Traversing the display list

As you've seen, the display list is a tree structure. At the top of the tree is the Stage, which can contain multiple display objects. Those display objects that are themselves display object containers can contain other display objects, or display object containers.



The DisplayObjectContainer class includes properties and methods for traversing the display list, by means of the child lists of display object containers. For example, consider the following code, which adds two display objects, `title` and `pict`, to the `container` object (which is a Sprite, and the Sprite class extends the DisplayObjectContainer class):

```
var container:Sprite = new Sprite();
var title:TextField = new TextField();
title.text = "Hello";
var pict:Loader = new Loader();
var url:URLRequest = new URLRequest("banana.jpg");
pict.load(url);
pict.name = "banana loader";
container.addChild(title);
container.addChild(pict);
```

The `getChildAt()` method returns the child of the display list at a specific index position:

```
trace(container.getChildAt(0) is TextField); // true
```

You can also access child objects by name. Each display object has a name property, and if you don't assign it, Flash Player assigns a default value, such as `"instance1"`. For example, the following code shows how to use the `getChildByName()` method to access a child display object with the name `"banana loader"`:

```
trace(container.getChildByName("banana loader") is Loader); // true
```

Using the `getChildByName()` method can result in slower performance than using the `getChildAt()` method.

Since a display object container can contain other display object containers as child objects in its display list, you can traverse the full display list of the application as a tree. For example, in the code excerpt shown earlier, once the load operation for the `pict` Loader object is complete, the `pict` object will have one child display object, which is the bitmap, loaded. To access this bitmap display object, you can write `pict.getChildAt(0)`. You can also write `container.getChildAt(0).getChildAt(0)` (since `container.getChildAt(0) == pict`).

The following function provides an indented `trace()` output of the display list from a display object container:

```
function traceDisplayList(container:DisplayObjectContainer,
                        indentString:String = ""):void
{
  var child:DisplayObject;
  for (var i:uint=0; i < container.numChildren; i++)
  {
    child = container.getChildAt(i);
    trace (indentString, child, child.name);
    if (container.getChildAt(i) is DisplayObjectContainer)
    {
      traceDisplayList(DisplayObjectContainer(child), indentString + "")
    }
  }
}
```

If you use Flex, you should know that Flex defines many component display object classes, and these classes override the display list access methods of the DisplayObjectContainer class. For example, the Container class of the mx.core package overrides the `addChild()` method and other methods of the DisplayObjectContainer class (which the Container class extends). In the case of the `addChild()` method, the class overrides the method in such a way that you cannot add all types of display objects to a Container instance in Flex. The overridden method, in this case, requires that the child object that you are adding be a type of mx.core.UIComponent object.

Setting Stage properties

The Stage class overrides most properties and methods of the DisplayObject class. If you call one of these overridden properties or methods, Flash Player throws an exception. For example, the Stage object does not have `x` or `y` properties, since its position is fixed as the main container for the application. The `x` and `y` properties refer to the position of a display object relative to its container, and since the Stage is not contained in another display object container, these properties do not apply.

NOTE

Some properties and methods of the Stage class are not available to display objects that are not in the same security sandbox as the first SWF file loaded. For details, see "Security sandboxes" on page 461.

Controlling the playback frame rate

The `framerate` property of the Stage class is used to set the frame rate for all SWF files loaded into the application. For more information, see the *ActionScript 3.0 Language Reference*.

Working with full-screen mode

Full-screen mode allows you to make a SWF fill a viewer's entire monitor, without any borders, menu bars, and so forth. The Stage class's `displayState` property is used to toggle full-screen mode on and off for a SWF. The `displayState` property can be set to one of the values defined by the constants in the flash.display.StageDisplayState class. To turn on full-screen mode, set `displayState` to `StageDisplayState.FULL_SCREEN`:

```
// mySprite is a Sprite instance, already added to the display list
mySprite.stage.displayState = StageDisplayState.FULL_SCREEN;
```

To exit full-screen mode, set the `displayState` property to `StageDisplayState.NORMAL`:

```
mySprite.stage.displayState = StageDisplayState.NORMAL;
```

In addition, a user can choose to leave full-screen mode by switching focus to a different window or by pressing one of several key combinations: the escape key (all platforms), ctrl-w (Windows), cmd-w (Mac), or alt-F4 (Windows).

Stage scaling behavior for full-screen mode is the same as under normal mode; the scaling is controlled by the Stage class's `scaleMode` property. As always, if the `scaleMode` property is set to `StageScaleMode.NO_SCALE`, the stage's `stageWidth` and `stageHeight` properties change to reflect the size of the screen area occupied by the SWF (the entire screen, in this case).

You can use the Stage class's `fullScreen` event to detect and respond when full-screen mode is turned on or off. For example, you might want to reposition, add, or remove items from the screen when entering or leaving full-screen mode, as in this example:

```
import flash.events.FullScreenEvent;

function fullScreenRedraw(event:FullScreenEvent):void
{
  if (event.fullScreen)
  {
    // remove input text fields
    // add a button which closes full-screen mode
  }
  else
  {
    // re-add input text fields
    // remove the button which closes full-screen mode
  }
}

mySprite.stage.addEventListener(FullScreenEvent.FULL_SCREEN,
  fullScreenRedraw);
```

As this code shows, the event object for the `fullScreen` event is an instance of the flash.events.FullScreenEvent class, which includes a `fullScreen` property indicating whether full-screen mode is enabled (`true`) or not (`false`).

When working with full-screen mode in ActionScript, you'll want to keep the following considerations in mind:

- Full-screen mode can only be initiated through ActionScript in response to a mouse click (including right-click) or keypress.
- For users with multiple monitors, the SWF content will only expand to fill one monitor. Flash Player uses a metric to determine which monitor contains the greatest portion of the SWF, and uses that monitor for full-screen mode.
- For a SWF file embedded in an HTML page, the HTML code to embed Flash Player must include a `<param>` tag and `<embed>` attribute with name `allowFullScreen` and value `true`, such as this:

```
<object>
    ...
    <param name="allowFullScreen" value="true" />
    <embed ... allowfullscreen="true" />
</object>
```

If you are using JavaScript in a web page to generate the SWF-embedding tags, you must alter the JavaScript to add the allowFullScreen param tag/attribute. For example, if your HTML page uses the `AC_FL_RunContent()` function (which is used by both Flex Builder- and Flash-generated HTML pages) you should add the allowFullScreen parameter to that function call as follows:

```
AC_FL_RunContent(
    ...
    'allowFullScreen','true',
    ...
    ); //end AC code
```

This does not apply to SWF files running in the standalone Flash Player.

- All keyboard-related ActionScript, such as keyboard events and text entry in TextFields, is disabled in full-screen mode. The exception is the keyboard shortcuts which close full-screen mode.

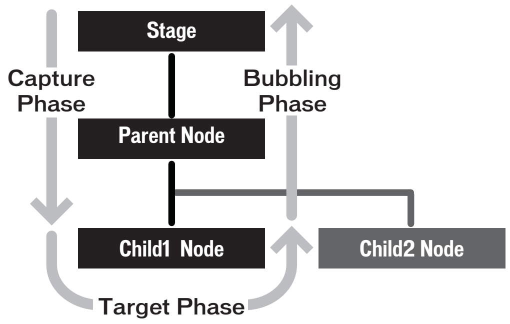
There are a few additional security-related restrictions you'll want to understand too. These are described in "Full-screen mode security" on page 465.

Handling events for display objects

The DisplayObject class inherits from the EventDispatcher class. This means that every display object can participate fully in the event model (described in Chapter 13, "Handling Events," on page 345). Every display object can use its `addEventListener()` method—inherited from the EventDispatcher class—to listen for a particular event, but only if the listening object is part of the event flow for that event.

When Flash Player dispatches an event object, that event object makes a round-trip journey from the Stage to the display object where the event occurred. For example, if a user clicks on a display object named `child1`, Flash Player dispatches an event object from the Stage through the display list hierarchy down to the `child1` display object.

The event flow is conceptually divided into three phases, as illustrated in this diagram:



For more information, see Chapter 13, "Handling Events," on page 345.

Basics for working with the core display classes

The sections that follow provide a starting point for working with the core display classes in ActionScript 3.0. These sections provide only an overview of these topics. For more details, see the sections describing the applicable classes in the *ActionScript 3.0 Language Reference*.

Drawing vector graphics

Each Shape, Sprite, and MovieClip object has a `graphics` property. The `graphics` property for each of these objects is a Graphics object, and the Graphics class includes properties and methods for drawing and manipulating lines, fills, and shapes.

For example, the following code draws an orange circle in a Shape object:

```
import flash.display.*;
var circle:Shape = new Shape()
var xPos:Number = 100;
var yPos:Number = 100;
var radius:Number = 50;
circle.graphics.beginFill(0xFF8800);
circle.graphics.drawCircle(xPos, yPos, radius);
this.addChild(circle);
```

The Graphics class includes the following methods for easily drawing simple shapes: `drawCircle()`, `drawEllipse()`, `drawRect()`, `drawRoundRect()`, and `drawRoundRectComplex()`. Before calling drawing methods, define the line style, fill, or both by calling the `linestyle()`, `lineGradientStyle()`, `beginFill()`, `beginGradientFill()`, or `beginBitmapFill()` method.

Use the Sprite class if you want to create a graphical object that is also a display object container (to contain other display objects) but that does not require a timeline.

For example, the following Sprite object has a circle drawn with its `graphics` property, and it has a TextField object in its child list:

```
var mySprite:Sprite = new Sprite();
mySprite.graphics.beginFill(0xFFCC00);
mySprite.graphics.drawCircle(30, 30, 30);
var label:TextField = new TextField();
label.text = "hello";
label.x = 20;
label.y = 20;
mySprite.addChild(label);
this.addChild(mySprite);
```

The graphics layer for a Sprite or MovieClip object always appears in back of the child display objects of the Sprite or MovieClip. Also, the graphics layer does not appear in the child list of the Sprite or MovieClip.

Working with text

The TextField class, which is in the flash.text package, lets you work with dynamic and input text fields. Note that there is also a StaticText class in the flash.text package. However, you cannot instantiate StaticText objects in ActionScript; these are created in the Flash authoring tool.

As the following example shows, you can use a TextFormat object to set formatting for an entire TextField or for a range of text:

```
var tf:TextField = new TextField();
tf.text = "Hello Hello";

var format1:TextFormat = new TextFormat();
format1.color = 0xFF0000;

var format2:TextFormat = new TextFormat();
format2.font = "Courier";

tf.setTextFormat(format1);
var startRange:uint = 6;
tf.setTextFormat(format2, startRange);

addChild(tf);
```

Text fields can contain either plain text or HTML-formatted text. Plain text is stored in the `text` property of the instance, and HTML text is stored in the `htmlText` property.

You can apply CSS style sheets to HTML text by using a StyleSheet object. For example:

```
var style:StyleSheet = new StyleSheet();

var styleObj:Object = new Object();
styleObj.fontSize = "bold";
styleObj.color = "#FF0000";
style.setStyle(".darkRed", styleObj);
delete styleObj;

var tf:TextField = new TextField();
tf.styleSheet = style;
tf.htmlText = "<span class = 'darkRed'>Red</span> apple";

addChild(tf);
```

For more information, see the TextField class in the *ActionScript 3.0 Language Reference*.

Loading content dynamically

You can load any of the following external display assets into an ActionScript 3.0 application:

- A SWF file authored in ActionScript 3.0—This file can be a Sprite, MovieClip, or any class that extends Sprite.
- An image file—This includes JPG, PNG, and GIF files.
- An AVM1 SWF file—This is a SWF file written in ActionScript 1.0 or 2.0.

You load these assets by using the Loader class.

The Loader class

Loader objects are used to load SWF files and graphics files into an application. The Loader class is a subclass of the DisplayObjectContainer class. A Loader object can contain only one child display object in its display list—the display object representing the SWF or graphic file that it loads. When you add a Loader object to the display list, as in the following code, you also add the loaded child display object to the display list once it loads:

```
var pictLdr:Loader = new Loader();
var pictURL:String = "banana.jpg"
var pictURLReq:URLRequest = new URLRequest(pictURL);
pictLdr.load(pictURLReq);
this.addChild(pictLdr);
```

Once the SWF file or image is loaded, you can move the loaded display object to another display object container, such as the `container` DisplayObjectContainer object in this example:

```
import flash.display.*;
import flash.net.URLRequest;
import flash.events.Event;
var container:Sprite = new Sprite();
addChild(container);
var pictLdr:Loader = new Loader();
var pictURL:String = "banana.jpg"
var pictURLReq:URLRequest = new URLRequest(pictURL);
pictLdr.load(pictURLReq);
pictLdr.contentLoaderInfo.addEventListener(Event.COMPLETE, imgLoaded);
function imgLoaded(e:Event):void
{
  container.addChild(pictLdr.content);
}
```

The LoaderInfo class

Once the file has loaded, a LoaderInfo object is created. This object is a property of both the Loader object and the loaded display object. The LoaderInfo object is a property of the Loader object through the `contentLoaderInfo` property of the Loader object. It is a property of the loaded display object through the display object's `loaderInfo` property. The `loaderInfo` property of the loaded display object refers to the same LoaderInfo object as the `contentLoaderInfo` property of the Loader object. In other words, a LoaderInfo object is shared between a loaded object and the Loader object that loaded it (between loader and loadee).

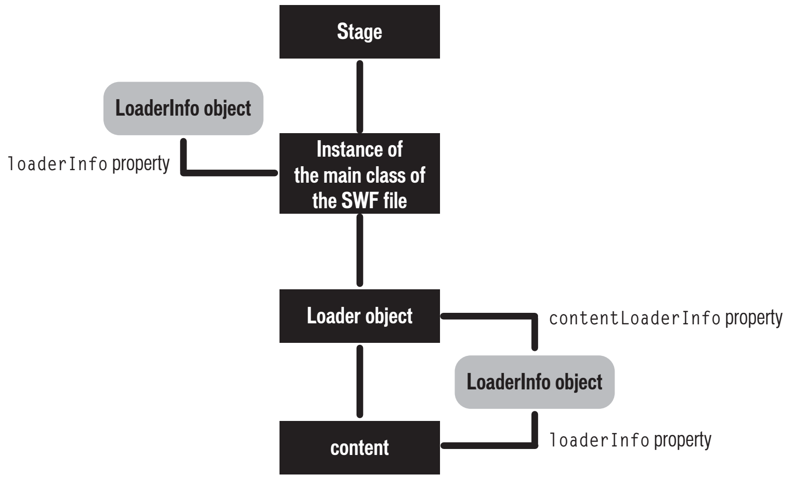
The LoaderInfo class provides information such as load progress, the URLs of the loader and loadee, the number of bytes total for the media, and the nominal height and width of the media. A LoaderInfo object also dispatches events for monitoring the progress of the load.

In order to access properties of loaded content, you will want to add an event listener to the LoaderInfo object, as in the following code:

```
import flash.display.Loader;
import flash.display.Sprite;
import flash.events.Event;

var ldr:Loader = new Loader();
var urlReq:URLRequest = new URLRequest("Circle.swf");
ldr.load(urlReq);
ldr.contentLoaderInfo.addEventListener(Event.COMPLETE, loaded);
addChild(ldr);

function loaded(event:Event):void
{
  var content:Sprite = event.target.content;
  content.scaleX = 2;
}
```

For more information, see Chapter 13, “Handling Events,” on page 345.

The following diagram shows the different uses of the LoaderInfo object—for the instance of the main class of the SWF file, for a Loader object, and for an object loaded by the Loader object:



The LoaderContext class

When you load an external file into Flash Player through the `load()` or `loadBytes()` method of the Loader class, you can optionally specify a `context` parameter. This parameter is a LoaderContext object.

The LoaderContext class includes three properties that let you define the context of how the loaded content can be used:

- `checkPolicyFile`—Use this property only when loading an image file (not a SWF file). If you set this property to `true`, the Loader checks the origin server for a cross-domain policy file (see "Website controls (cross-domain policy files)" on page 456). This is necessary only for content originating from domains other than that of the SWF file containing the Loader object. If the server grants permission to the Loader domain, ActionScript from SWF files in the Loader domain can access data in the loaded image; in other words, you can use the `BitmapData.draw()` command to access data in the loaded image.

  Note that a SWF file from other domains than that of the Loader object can call `Security.allowDomain()` to permit a specific domain.

- `securityDomain`—Use this property only when loading a SWF file (not an image). Specify this for a SWF file from a domain other than that of the file containing the Loader object. When you specify this option, Flash Player checks for the existence of a cross-domain policy file, and if one exists, SWF files from the domains permitted in the cross-policy file can cross-script the loaded SWF content. You can specify `flash.system.SecurityDomain.currentDomain` as this parameter.
- `applicationDomain`—Use this property only when loading a SWF file written in ActionScript 3.0 (not an image or a SWF file written in ActionScript 1.0 or 2.0). When loading the file, you can specify that the file be included in the same application domain as that of the Loader object, by setting the `applicationDomain` parameter to `flash.system.ApplicationDomain.currentDomain`. By putting the loaded SWF file in the same application domain, you can access its classes directly. This can be useful if you are loading a SWF file that contains embedded media, which you can access via their associated class names. For more information, see "ApplicationDomain class" on page 436.

Here's an example of checking for a cross-domain policy file when loading a bitmap from another domain:

```
var context:LoaderContext = new LoaderContext();
context.checkPolicyFile = true;
var urlReq:URLRequest = new URLRequest("http://www.[your_domain_here].com/
  photo11.jpg");
var ldr:Loader = new Loader();
ldr.load(urlReq, context);
```

Here's an example of checking for a cross-domain policy file when loading a SWF from another domain, in order to place the file in the same security sandbox as the Loader object. Additionally, the code adds the classes in the loaded SWF file to the same application domain as that of the Loader object:

```
var context:LoaderContext = new LoaderContext();
context.securityDomain = SecurityDomain.currentDomain;
context.applicationDomain = ApplicationDomain.currentDomain;
var urlReq:URLRequest = new URLRequest("http://www.[your_domain_here].com/
  library.swf");
var ldr:Loader = new Loader();
ldr.load(urlReq, context);
```

For more information, see the LoaderContext class in the *ActionScript 3.0 Language Reference*.

Controlling ActionScript 3.0 movie clips

Each MovieClip object has a timeline, and also includes properties and methods for controlling the playhead and working with frames and scenes.

Many properties and methods of the ActionScript 1.0 and 2.0 MovieClip class are present as properties or methods of the ActionScript 3.0 MovieClip class. Some are inherited from a superclass. For example, the `x`, `y`, and `blendMode` properties of the DisplayObject class, to name a few, are valid for the MovieClip class because it extends the DisplayObject class. However, some properties that had names that begin with the underscore character, such as `_x`, `_y`, `_root`, have been renamed without the underscore in ActionScript 3.0. Also, the `_xmouse`, `_xscale`, `_ymouse`, and `_yscale` properties have been renamed `mouseX`, `scaleX`, `mouseY`, and `scaleY`.

The MovieClip class includes methods and properties for controlling the movie clip. The following properties let you monitor the total number of frames and scenes in a movie clip and monitor information about the frame and scene at the current playhead position: `currentFrame`, `currentLabel`, `currentLabels`, `currentScene`, `scenes`, and `totalFrames`. The following methods let you control the playhead for the movie clip: `gotoAndPlay()`, `gotoAndStop()`, `nextFrame()`, `nextScene()`, `play()`, `prevFrame()`, `prevScene()`, and `stop()`.

For example, the following code moves the playhead to the first frame of the current scene of the `myMovie` MovieClip object:

```
myMovie.gotoAndPlay(0);
```

For more information on these and other properties and methods, see the MovieClip section of the *ActionScript 3.0 Language Reference.*

You can set the frame rate for all movie clips in an application using the `frameRate` property of the Stage object.

In contrast to ActionScript 1.0 and 2.0, a loaded SWF file in ActionScript 3.0 can be a MovieClip object, an AVM1Movie object, or a Sprite object. However, in ActionScript 3.0 you can control the timeline only of a MovieClip object. Sprite and AVM1Movie objects do not include timeline-related methods and properties.

Creating and manipulating bitmaps

The BitmapData class lets you manipulate the pixels of a Bitmap object. This can be a bitmap that you loaded from a file, or one that you draw exclusively through one of the BitmapData methods. Each Bitmap object has a `bitmapData` property that is a BitmapData object.

A BitmapData object represents a rectangular array of pixels. You can create a new bitmap programmatically. The BitmapData constructor lets you create a rectangle with a specified color, and you can attach that BitmapData object to a new Bitmap object, as shown in the following example:

```
var bdWidth:Number = 100;
var bdHeight:Number = 100;
var bdTransparent:Boolean = true;
var bdFillColorARGB:uint = 0xFF007090;
var myBitmapData:BitmapData = new BitmapData(bdWidth,
         bdHeight,
         bdTransparent,
         bdFillColorARGB);
var myBitmap:Bitmap = new Bitmap(myBitmapData);
addChild(myBitmap)
```

More commonly, however, you will work with bitmap data from a loaded image file. In either case, the BitmapData class includes methods for working with and modifying the BitmapData. For example, you can use the `setPixel()` method to set a pixel to a specific RGB value. For details on properties and methods of the Bitmap and BitmapData class, see the *ActionScript 3.0 Language Reference*.

Note that the `bitmapData` property of multiple Bitmap objects can reference the same BitmapData object, and you can apply different effects and transformations to each Bitmap object.

Working with SimpleButton objects

In ActionScript 3.0, you can define button behavior through the SimpleButton class. A SimpleButton has three states: `upState`, `downState`, and `overState`. These are each properties of the SimpleButton object, and they are each DisplayObject objects. For example, the following class defines a simple text button:

```
import flash.display.*;
import flash.events.*;

public class TextButton extends SimpleButton
{
  public var selected:Boolean = false;
  public function TextButton(txt:String)
  {
    upState = new TextButtonState(0xFFFFFF, txt);
    downState = new TextButtonState(0xCCCCCC, txt);
    hitTestState = upState;
    overState = upState;
    addEventListener(MouseEvent.CLICK, buttonClicked);
  }
  public function buttonClicked(e:Event)
  {
    trace("Button clicked.");
  }
}
```

The `hitTestState` property of a SimpleButton object is a DisplayObject instance that responds to the mouse events for the button. In this example, we set the `hitTestState` property (and the `overState` property) to be the same DisplayObject instance as the `upState` property. The TextButton class in the example references the following class, which defines the DisplayObject to be used for a button state (up, down, or over):

```
import flash.display.*;
import flash.text.TextFormat;
import flash.text.TextField;

class TextButtonState extends Sprite
{
  public var label:TextField;
  public function TextButtonState(color:uint, labelText:String)
  {
    label = new TextField();
    label.text = labelText;
    label.x = 2;
    var format:TextFormat = new TextFormat("Verdana");
    label.setTextFormat(format);
    var buttonWidth:Number = label.textWidth + 10;
    var background:Shape = new Shape();
    background.graphics.beginFill(color);
    background.graphics.lineStyle(1, 0x000000);
    background.graphics.drawRoundRect(0, 0, buttonWidth, 18, 4);
    addChild(background);
    addChild(label);
  }
}
```

Working with video

The Video class is not in the flash.display package, but it is a subclass of the DisplayObject class. To have a video attached to the Video object, you must use the `attachNetStream()` method or the `attachCamera()` method.

Here is a simple example that attaches a net stream to a video and adds the video to the Sprite display object container:

```
import flash.display.Sprite;
import flash.net.*;
import flash.media.Video;

public class VideoTest extends Sprite
{
  private var videoUrl:String = "http://example.com/test.flv";

  public function VideoTest()
  {
    var connection:NetConnection = new NetConnection();
    connection.connect(null);

    var stream:NetStream = new NetStream(connection);

    var myVideo:Video = new Video(360, 240);
    myVideo.attachNetStream(stream);
    stream.play(videoUrl);

    addChild(myVideo);
  }
}
```

For more information, see the Video class in the *ActionScript 3.0 Language Reference*.

Example: SpriteArranger

The SpriteArranger example application builds upon the Geometric Shapes example application described in Chapter 4 (see "Example: GeometricShapes" on page 148).

The SpriteArranger example application illustrates a number of concepts for dealing with display objects:

- Extending display object classes
- Adding objects to the display list
- Layering display objects and working with display object containers
- Responding to display object events
- Using properties and methods of display objects

The SpriteArranger application files can be found in the folder Examples/SpriteArranger. The application consists of the following files:

File	Description
SpriteArranger.mxml	The main application file in MXML for Flex.
com/example/programmingas3/ SpriteArranger/CircleSprite.as	A class defining a type of Sprite object that renders a circle onscreen.
com/example/programmingas3/ SpriteArranger/DrawingCanvas.as	A class defining the canvas, which is a display object container that contains GeometricSprite objects.
com/example/programmingas3/ SpriteArranger/SquareSprite.as	A class defining a type of Sprite object that renders a square onscreen.
com/example/programmingas3/ SpriteArranger/TriangleSprite.as	A class defining a type of Sprite object that renders a triangle onscreen.
com/example/programmingas3/ SpriteArranger/GeometricSprite.as	A class that extends the Sprite object, used to define an onscreen shape. The CircleSprite, SquareSprite, and TriangleSprite each extend this class.
com/example/programmingas3/ geometricshapes/IGeometricShape.as	The base interface defining methods to be implemented by all geometric shape classes.
com/example/programmingas3/ geometricshapes/IPolygon.as	An interface defining methods to be implemented by geometric shape classes that have multiple sides.
com/example/programmingas3/ geometricshapes/RegularPolygon.as	A type of geometric shape that has sides of equal length postioned symmetrically around the shape's center.
com/example/programmingas3/ geometricshapes/Circle.as	A type of geometric shape that defines a circle.

File	Description
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	A subclass of RegularPolygon that defines a triangle with all sides the same length.
com/example/programmingas3/geometricshapes/Square.as	A subclass of RegularPolygon defining a rectangle with all four sides the same length.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	A class containing a "factory method" for creating shapes given a shape type and size.

Defining the SpriteArranger classes

The SpriteArranger application lets the user add a variety of display objects to the onscreen "canvas."

The DrawingCanvas class defines a drawing area, a type of display object container, to which the user can add onscreen shapes. These onscreen shapes are instances of one of the subclasses of the GeometricSprite class.

The DrawingCanvas class

In Flex, all child display objects added to a Container object must be of a class that descends from the mx.core.UIComponent class. This application adds an instance of the DrawingCanvas class as a child of an mx.containers.VBox object, as defined in MXML code in the SpriteArranger.mxml file. This inheritance is defined in the DrawingCanvas class declaration, as follows:

```
public class DrawingCanvas extends UIComponent
```

The UIComponent class inherits from the DisplayObject, DisplayObjectContainer, and Sprite classes, and the code in the DrawingCanvas class uses methods and properties of those classes.

The `DrawingCanvas()` constructor method sets up a Rectangle object, `bounds`, which is property that is later used in drawing the outline of the canvas. It then calls the `initCanvas()` method, as follows:

```
this.bounds = new Rectangle(0, 0, w, h);
initCanvas(fillColor, lineColor);
```

AS the following example shows, the `initCanvas()` method defines various properties of the DrawingCanvas object, which were passed as arguments to the constructor function:

```
this.lineColor = lineColor;
this.fillColor = fillColor;
this.width = 500;
this.height = 200;
```

The `initCanvas()` method then calls the `drawBounds()` method, which draws the canvas using the DrawingCanvas class's `graphics` property. The `graphics` property is inherited from the Shape class.

```
this.graphics.clear();
this.graphics.lineStyle(1.0, this.lineColor, 1.0);
this.graphics.beginFill(this.fillColor, 1.0);
this.graphics.drawRect(bounds.left - 1,
              bounds.top - 1,
              bounds.width + 2,
              bounds.height + 2);
this.graphics.endFill();
```

The following additional methods of the DrawingCanvas class are invoked based on user interactions with the application:

- The `addShape()` and `describeChildren()` methods, which are described in "Adding display objects to the canvas" on page 191.
- The `moveToBack()`, `moveDown()`, `moveToFront()`, and `moveUp()` methods, which are described in "Rearranging display object layering" on page 194.
- The `onMouseUp()` method, which is described in "Clicking and dragging display objects" on page 192.

The GeometricSprite class and its subclasses

Each display object the user can add to the canvas is an instance of one of the following subclasses of the GeometricSprite class:

- CircleSprite
- SquareSprite
- TriangleSprite

The GeometricSprite class extends the flash.display.Sprite class:

```
public class GeometricSprite extends Sprite
```

The GeometricSprite class includes a number of properties common to all GeometricSprite objects. These are set in the constructor function based on parameters passed to the function. For example:

```
this.size = size;
this.lineColor = lColor;
this.fillColor = fColor;
```

The `geometricShape` property of the GeometricSprite class defines an IGeometricShape interface, which defines the mathematical properties, but not the visual properties, of the shape. The classes that implement the IGeometricShape interface are defined in the example application for Chapter 4 (see "Example: GeometricShapes" on page 148).

The GeometricSprite class defines the `drawShape()` method, which is further refined in the override definitions in each subclass of GeometricSprite. For more information, see the "Adding display objects to the canvas" section, which follows.

The GeometricSprite class also provides the following methods:

- The `onMouseDown()` and `onMouseUp()` methods, which are described in "Clicking and dragging display objects" on page 192.
- The `showSelected()` and `hideSelected()` methods, which are described in "Clicking and dragging display objects" on page 192.

Adding display objects to the canvas

When the user clicks the Add Shape button, the application calls the `addShape()` method of the DrawingCanvas class. It instantiates a new GeometricSprite by calling the appropriate constructor function of one of the GeometricSprite subclasses, as the following example shows:

```
public function addShape(shapeName:String, len:Number):void
{
  var newShape:GeometricSprite;
  switch (shapeName)
  {
    case "Triangle":
      newShape = new TriangleSprite(len);
      break;
    case "Square":
      newShape = new SquareSprite(len);
      break;
    case "Circle":
      newShape = new CircleSprite(len);
      break;
  }
  newShape.alpha = 0.8;
  this.addChild(newShape);
}
```

Each constructor method calls the `drawShape()` method, which uses the `graphics` property of the class (inherited from the Sprite class) to draw the appropriate vector graphic. For example, the `drawShape()` method of the CircleSprite class includes the following code:

```
this.graphics.clear();
this.graphics.lineStyle(1.0, this.lineColor, 1.0);
this.graphics.beginFill(this.fillColor, 1.0);
var radius:Number = this.size / 2;
this.graphics.drawCircle(radius, radius, radius);
```

The second to last line of the `addShape()` function sets the `alpha` property of the display object (inherited from the DisplayObject class), so that each display object added to the canvas is slightly transparent, letting the user see the objects behind it.

The final line of the `addChild()` method adds the new display object to the child list of the instance of the DrawingCanvas class, which is already on the display list. This causes the new display object to appear on the Stage.

The interface for the application includes two text fields, `selectedSpriteTxt` and `outputTxt`. The text properties of these text fields are updated with information about the GeometricSprite objects that have been added to the canvas or selected by the user. The GeometricSprite class handles this information-reporting task by overriding the `toString()` method, as follows:

```
public override function toString():String
{
  return this.shapeType + " of size " + this.size + " at " + this.x + ", "
  + this.y;
}
```

The `shapeType` property is set to the appropriate value in the constructor method of each GeometricSprite subclass. For example, the `toString()` method might return the following value for a CircleSprite instance recently added to the DrawingCanvas instance:

```
Circle of size 50 at 0, 0
```

The `describeChildren()` method of the DrawingCanvas class loops through the canvas's child list, using the `numChildren` property (inherited from the DisplayObjectContainer class) to set the limit of the `for` loop. It generates a string listing each child, as follows:

```
var desc:String = "";
var child:DisplayObject;
for (var i:int = 0; i < this.numChildren; i++)
{
    child = this.getChildAt(i);
    desc += i + ": " + child + '\n';
}
```

The resulting string is used to set the `text` property of the `outputTxt` text field.

Clicking and dragging display objects

When the user clicks on a GeometricSprite instance, the application calls the `onMouseDown()` event handler. As the following shows, this event handler is set to listen for mouse down events in the constructor function of the GeometricSprite class:

```
this.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
```

The `onMouseDown()` method then calls the `showSelected()` method of the GeometricSprite object. If it is the first time this method has been called for the object, the method creates a new Shape object named `selectionIndicator` and it uses the `graphics` property of the Shape object to draw a red highlight rectangle, as follows:

```
this.selectionIndicator = new Shape();
this.selectionIndicator.graphics.lineStyle(1.0, 0xFF0000, 1.0);
this.selectionIndicator.graphics.drawRect(-1, -1, this.size + 1,
  this.size + 1);
this.addChild(this.selectionIndicator);
```

If this is not the first time the `onMouseDown()` method is called, the method simply sets the `selectionIndicator` shape's `visible` property (inherited from the DisplayObject class), as follows:

```
this.selectionIndicator.visible = true;
```

The `hideSelected()` method hides the `selectionIndicator` shape of the previously selected object by setting its `visible` property to `false`.

The `onMouseDown()` event handler method also calls the `startDrag()` method (inherited from the Sprite class), which includes the following code:

```
var boundsRect:Rectangle = this.parent.getRect(this.parent);
boundsRect.width -= this.size;
boundsRect.height -= this.size;
this.startDrag(false, boundsRect);
```

This lets the user drag the selected object around the canvas, within the boundaries set by the `boundsRect` rectangle.

When the user releases the mouse button, the `mouseUp` event is dispatched. The constructor method of the DrawingCanvas sets up the following event listener:

```
this.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
```

This event listener is set for the DrawingCanvas object, rather than for the individual GeometricSprite objects. This is because when the GeometricSprite object is dragged, it could end up behind another display object (another GeometricSprite object) when the mouse is released. The display object in the foreground would receive the mouse up event but the display object the user is dragging would not. Adding the listener to the DrawingCanvas object ensures that the event is always handled.

The `onMouseUp()` method calls the `onMouseUp()` method of the GeometricSprite object, which in turn calls the `stopDrag()` method of the GeometricSprite object.

Rearranging display object layering

The user interface for the application includes buttons labeled Move Back, Move Down, Move Up, and Move to Front. When the user clicks one of these buttons, the application calls the corresponding method of the DrawingCanvas class: `moveToBack()`, `moveDown()`, `moveUp()`, or `moveToFront()`. For example, the `moveToBack()` method includes the following code:

```
public function moveToBack(shape:GeometricSprite):void
{
    var index:int = this.getChildIndex(shape);
    if (index > 0)
    {
        this.setChildIndex(shape, 0);
    }
}
```

The method uses the `setChildIndex()` method (inherited from the DisplayObjectContainer class) to position the display object in index position 0 in the child list of the DrawingCanvas instance (`this`).

The `moveDown()` method works similarly, except that it decrements the index position of the display object by 1 in the child list of the DrawingCanvas instance:

```
public function moveDown(shape:GeometricSprite):void
{
    var index:int = this.getChildIndex(shape);
    if (index > 0)
    {
        this.setChildIndex(shape, index - 1);
    }
}
```

The `moveUp()` and `moveToFront()` methods work similarly to the `moveToBack()` and `moveDown()` methods.

PART 2

Core ActionScript 3.0 Data Types and Classes

2

This part describes the implementation of key ActionScript 3.0 classes and data types, and provides strategies for using them.

The following chapters are included:

CHAPTER 6

Working with Dates and Times

6

Timing might not be everything, but it's usually a key factor in software applications. ActionScript 3.0 provides powerful ways to manage calendar dates, times, and time intervals. Two main classes provide most of this timing functionality: the Date class and the new Timer class in the flash.utils package.

Contents

Managing calendar dates and times

All of the calendar date and time management functions in ActionScript 3.0 are concentrated in the top-level Date class. The Date class contains methods and properties that let you handle dates and times in either Coordinated Universal Time (UTC) or in local time specific to a time zone. UTC is a standard time definition that is essentially the same as Greenwich Mean Time (GMT).

Creating Date objects

The Date class boasts one of the most versatile constructor methods of all the core classes. You can invoke it four different ways.

First, if given no parameters, the `Date()` constructor returns a Date object containing the current date and time, in local time based on your time zone. Here’s an example:

```
var now:Date = new Date();
```

Second, if given a single numeric parameter, the `Date()` constructor treats that as the number of milliseconds since January 1, 1970, and returns a corresponding Date object. Note that the millisecond value you pass in is treated as milliseconds since January 1, 1970, in UTC. However, the Date object shows values in your local time zone, unless you use the UTC-specific methods to retrieve and display them. If you create a new Date object using a single milliseconds parameter, make sure you account for the time zone difference between your local time and UTC. The following statements create a Date object set to midnight on the day of January 1, 1970, in UTC:

```
var millisecondsPerDay:int = 1000 * 60 * 60 * 24;
// gets a Date one day after the start date of 1/1/1970
var startTime:Date = new Date(millisecondsPerDay);
```

Third, you can pass multiple numeric parameters to the `Date()` constructor. It treats those parameters as the year, month, day, hour, minute, second, and millisecond, respectively, and returns a corresponding Date object. Those input parameters are assumed to be in local time rather than UTC. The following statements get a Date object set to midnight at the start of January 1, 2000, in local time:

```
var millenium:Date = new Date(2000, 0, 1, 0, 0, 0, 0);
```

Fourth, you can pass a single string parameter to the `Date()` constructor. It will try to parse that string into date or time components and then return a corresponding Date object. If you use this approach, it's a good idea to enclose the `Date()` constructor in a `try..catch` block to trap any parsing errors. The `Date()` constructor accepts a number of different string formats, as listed in the *ActionScript 3.0 Language Reference*. The following statement initializes a new Date object using a string value:

```
var nextDay:Date = new Date("Mon May 1 2006 11:30:00 AM");
```

If the `Date()` constructor cannot successfully parse the string parameter, it will not raise an exception. However, the resulting Date object will contain an invalid date value.

Getting time unit values

You can extract the values for various units of time within a Date object using properties or methods of the Date class. Each of the following properties gives you the value of a time unit in the Date object:

- The `fullYear` property
- The `month` property, which is in a numeric format with 0 for January up to 11 for December
- The `date` property, which is the calendar number of the day of the month, in the range of 1 to 31

- The `day` property, which is the day of the week in numeric format, with 0 standing for Sunday
- The `hours` property, in the range of 0 to 23
- The `minutes` property
- The `seconds` property
- The `milliseconds` property

In fact, the Date class gives you a number of ways to get each of these values. For example, you can get the month value of a Date object in four different ways:

- The `month` property
- The `getMonth()` method
- The `monthUTC` property
- The `getMonthUTC()` method

All four ways are essentially equivalent in terms of efficiency, so you can use whichever approach suits your application best.

The properties just listed all represent components of the total date value. For example, the milliseconds property will never be greater than 999, since when it reaches 1000 the seconds value increases by 1 and the milliseconds property resets to 0.

If you want to get the value of the Date object in terms of milliseconds since January 1, 1970 (UTC), you can use the `getTime()` method. Its counterpart, the `setTime()` method, lets you change the value of an existing Date object using milliseconds since January 1, 1970 (UTC).

Performing date and time arithmetic

You can perform addition and subtraction on dates and times with the Date class. Date values are kept internally in terms of milliseconds, so you should convert other values to milliseconds before adding them to or subtracting them from Date objects.

If your application will perform a lot of date and time arithmetic, you might find it useful to create constants that hold common time unit values in terms of milliseconds, like the following:

```
public static const millisecondsPerMinute:int = 1000 * 60;
public static const millisecondsPerHour:int = 1000 * 60 * 60;
public static const millisecondsPerDay:int = 1000 * 60 * 60 * 24;
```

Now it is easy to perform date arithmetic using standard time units. The following code sets a date value to one hour from the current time using the `getTime()` and `setTime()` methods:

```
var oneHourFromNow:Date = new Date();
oneHourFromNow.setTime(oneHourFromNow.getTime() + millisecondsPerHour);
```

Another way to set a date value is to create a new Date object using a single milliseconds parameter. For example, the following code adds 30 days to one date to calculate another:

```
// sets the invoice date to today's date
var invoiceDate:Date = new Date();

// adds 30 days to get the due date
var dueDate:Date = new Date(invoiceDate.getTime() + (30 *
  millisecondsPerDay));
```

Next, the `millisecondsPerDay` constant is multiplied by 30 to represent 30 days' time and the result is added to the `invoiceDate` value and used to set the `dueDate` value.

Converting between time zones

Date and time arithmetic comes in handy when you want to convert dates from one time zone to another. So does the `getTimezoneOffset()` method, which returns the value in minutes by which the Date object's time zone differs from UTC. It returns a value in minutes because not all time zones are set to even-hour increments—some have half-hour offsets from neighboring zones.

The following example uses the time zone offset to convert a date from local time to UTC. It does the conversion by first calculating the time zone value in milliseconds and then adjusting the Date value by that amount:

```
// creates a Date in local time
var nextDay:Date = new Date("Mon May 1 2006 11:30:00 AM");

// converts the Date to UTC by adding or subtracting the time zone offset
var offsetMilliseconds:Number = nextDay.getTimezoneOffset() * 60 * 1000;
nextDay.setTime(nextDay.getTime() + offsetMilliseconds);
```

Controlling time intervals

When you develop applications using the Flash authoring tool, you have access to the timeline, which provides a steady, frame-by-frame progression through your application. In pure ActionScript projects, however, you must rely on other timing mechanisms.

Loops versus timers

In some programming languages, you must devise your own timing schemes using loop statements like `for` or `do..while`.

Loop statements generally execute as fast as the local machine allows, which means that the application runs faster on some machines and slower on others. If your application needs a consistent timing interval, you need to tie it to an actual calendar or clock time. Many applications, such as games, animations, and real-time controllers, need regular, time-driven ticking mechanisms that are consistent from machine to machine.

The ActionScript 3.0 Timer class provides a powerful solution. Using the ActionScript 3.0 event model, the Timer class dispatches timer events whenever a specified time interval is reached.

The Timer class

The preferred way to handle timing functions in ActionScript 3.0 is to use the Timer class (flash.utils.Timer), which can be used to dispatch events whenever an interval is reached.

To start a timer, you first create an instance of the Timer class, telling it how often to generate a timer event and how many times to do so before stopping.

For example, the following code creates a Timer instance that dispatches an event every second and continues for 60 seconds:

```
var oneMinuteTimer:Timer = new Timer(1000, 60);
```

The Timer object dispatches a TimerEvent object each time the given interval is reached. A TimerEvent object's event type is `timer` (defined by the constant `TimerEvent.TIMER`). A TimerEvent object contains the same properties as a standard Event object.

If the Timer instance is set to a fixed number of intervals, it will also dispatch a `timerComplete` event (defined by the constant `TimerEvent.TIMER_COMPLETE`) when it reaches the final interval.

Here is a small sample application showing the Timer class in action:

```
package
{
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.utils.Timer;

  public class ShortTimer extends Sprite
  {
    public function ShortTimer()
    {
      // creates a new five-second Timer
      var minuteTimer:Timer = new Timer(1000, 5);

      // designates listeners for the interval and completion events
      minuteTimer.addEventListener(TimerEvent.TIMER, onTick);
      minuteTimer.addEventListener(TimerEvent.TIMER_COMPLETE,
  onTimerComplete);

      // starts the timer ticking
      minuteTimer.start();
    }

    public function onTick(evt:TimerEvent):void
    {
      // displays the tick count so far
      // The target of this event is the Timer instance itself.
      trace("tick " + evt.target.currentCount);
    }

    public function onTimerComplete(evt:TimerEvent):void
    {
      trace("Time's Up!");
    }
  }
}
```

When the ShortTimer class is created, it creates a Timer instance that will tick once per second for five seconds. Then it adds two listeners to the timer: one that listens to each tick, and one that listens for the `timerComplete` event.

Next, it starts the timer ticking, and from that point forward, the `onTick()` method executes at one-second intervals.

The `onTick()` method simply displays the current tick count. After five seconds have passed, the `onTimerComplete()` method executes, telling you that the time is up.

When you run this sample, you should see the following lines appear in your console or trace window at the rate of one line per second:

```
tick 1
tick 2
tick 3
tick 4
tick 5
Time's Up!
```

Timing functions in the flash.utils package

ActionScript 3.0 contains a number of timing functions similar to those that were available in ActionScript 2.0. These functions are provided as package-level functions in the flash.utils package, and they operate just as they did in ActionScript 2.0.

Function	Description
`clearInterval(id:uint):void`	Cancels a specified `setInterval()` call.
`clearTimeout(id:uint):void`	Cancels a specified `setTimeout()` call.
`getTimer():int`	Returns the number of milliseconds that have elapsed since the Flash Player was initialized.
`setInterval(closure:Function, delay:Number, ... arguments):uint`	Runs a function at a specified interval (in milliseconds).
`setTimeout(closure:Function, delay:Number, ... arguments):uint`	Runs a specified function after a specified delay (in milliseconds).

These functions remain in ActionScript 3.0 for backward compatibility. Adobe does not recommend that you use them in new ActionScript 3.0 applications. In general, it is easier and more efficient to use the Timer class in your applications.

Example: Simple analog clock

A simple analog clock example illustrates two of the date and time concepts discussed in this chapter:

- Getting the current date and time and extracting values for the hours, minutes, and seconds
- Using a Timer to set the pace of an application

The SimpleClock application files can be found in the folder Samples/SimpleClock. The application consists of the following files:

File	Description
SimpleClockApp.mxml	The main application file in MXML for Flex.
SimpleClockApp.fla	The main application file as a FLA file for the Flash authoring tool.
com/example/programmingas3/simpleclock/SimpleClock.as	The main application file.
com/example/programmingas3/simpleclock/AnalogClockFace.as	Draws a round clock face and hour, minute, and seconds hands based on the time.

Defining the SimpleClock class

The clock example is simple, but it's a good idea to organize even simple applications well so you could easily expand them in the future. To that end, the SimpleClock application uses the SimpleClock class to handle the startup and time-keeping tasks, and then uses another class named AnalogClockFace to actually display the time.

Here is the code that defines and initializes the SimpleClock class:

```
public class SimpleClock extends UIComponent
{
  /**
   * The time display component.
   */
  private var face:AnalogClockFace;

  /**
   * The Timer that acts like a heartbeat for the application.
   */
  private var ticker:Timer;
```

The class has two important properties:

- The `face` property, which is an instance of the AnalogClockFace class
- The `ticker` property, which is an instance of the Timer class

The SimpleClock class uses a default constructor. The `initClock()` method takes care of the real setup work, creating the clock face and starting the Timer instance ticking.

Creating the clock face

The next lines in the SimpleClock code create the clock face that is used to display the time:

```
/**
 * Sets up a SimpleClock instance.
 */
public function initClock(faceSize:Number = 200)
{
  // creates the clock face and adds it to the display list
  face = new AnalogClockFace(Math.max(20, faceSize));
  face.init();
  addChild(face);

  // draws the initial clock display
  face.draw();
```

The size of the face can be passed in to the `initClock()` method. If no `faceSize` value is passed, a default size of 200 pixels is used.

Next, the application initializes the face and then adds it to the display list using the `addChild()` method inherited from the DisplayObject class. Then it calls the `AnalogClockFace.draw()` method to display the clock face once, showing the current time.

Starting the timer

After creating the clock face, the `initClock()` method sets up a timer:

```
    // creates a Timer that fires an event once per second
    ticker = new Timer(1000);

    // designates the onTick() method to handle Timer events
    ticker.addEventListener(TimerEvent.TIMER, onTick);

    // starts the clock ticking
    ticker.start();
```

First this method instantiates a Timer instance that will dispatch an event once per second (every 1000 milliseconds). Since no second `repeatCount` parameter is passed to the `Timer()` constructor, the Timer will keep repeating indefinitely.

The `SimpleClock.onTick()` method will execute once per second when the `timer` event is received:

```
  public function onTick(evt:TimerEvent):void
  {
    // updates the clock display
    face.draw();
  }
```

The `AnalogClockFace.draw()` method simply draws the clock face and hands.

Displaying the current time

Most of the code in the AnalogClockFace class involves setting up the clock face's display elements. When the AnalogClockFace is initialized, it draws a circular outline, places a numeric text label at each hour mark, and then creates three Shape objects, one each for the hour hand, the minute hand, and the second hand on the clock.

Once the SimpleClock application is running, it calls the `AnalogClockFace.draw()` method each second, as follows:

```
  /**
   * Called by the parent container when the display is being drawn.
   */
  public override function draw():void
  {
    // stores the current date and time in an instance variable
    currentTime = new Date();
    showTime(currentTime);
  }
```

This method saves the current time in a variable, so the time can't change in the middle of drawing the clock hands. Then it calls the `showTime()` method to display the hands, as the following shows:

```
/**
 * Displays the given Date/Time in that good old analog clock style.
 */
public function showTime(time:Date):void
{
    // gets the time values
    var seconds:uint = time.getSeconds();
    var minutes:uint = time.getMinutes();
    var hours:uint = time.getHours();

    // multiplies by 6 to get degrees
    this.secondHand.rotation = 180 + (seconds * 6);
    this.minuteHand.rotation = 180 + (minutes * 6);

    // Multiply by 30 to get basic degrees, then
    // add up to 29.5 degrees (59 * 0.5)
    // to account for the minutes.
    this.hourHand.rotation = 180 + (hours * 30) + (minutes * 0.5);
}
```

First, this method extracts the values for the hours, minutes, and seconds of the current time. Then it uses these values to calculate the angle for each hand. Since the second hand makes a full rotation in 60 seconds, it rotates 6 degrees each second (360/60). The minute hand rotates the same amount each minute.

The hour hand updates every minute, too, so it can show some progress as the minutes tick by. It rotates 30 degrees each hour (360/12), but it also rotates half a degree each minute (30 degrees divided by 60 minutes).

CHAPTER 7

Working with Strings

7

The String class contains methods that let you work with text strings. Strings are important in working with many objects. The methods described in this chapter are useful in working with strings used in objects such as TextField, StaticText, XML, ContextMenu, and FileReference objects.

Strings are sequences of characters. ActionScript 3.0 supports ASCII and Unicode characters.

Contents

Creating strings

The String class is used to represent string (textual) data in ActionScript 3.0. ActionScript strings support both ASCII and Unicode characters. The simplest way to create a string is to use a string literal. To declare a string literal, use straight double quotation mark (`"`) or single quotation mark (`'`) characters. For example, the following two strings are equivalent:

```
var str1:String = "hello";
var str2:String = 'hello';
```

You can also declare a string by using the `new` operator, as follows:

```
var str1:String = new String("hello");
var str2:String = new String(str1);
var str3:String = new String();       // str3 == null
```

The following two strings are equivalent:

```
var str1:String = "hello";
var str2:String = new String("hello");
```

To use single quotation marks (`'`) within a string literal defined with single quotation mark (`'`) delimiters, use the backslash escape character (`\`). Similarly, to use double quotation marks (`"`) within a string literal defined with double quotation marks (`"`) delimiters, use the backslash escape character (`\`). The following two strings are equivalent:

```
var str1:String = "That's \"A-OK\"";
var str2:String = 'That\'s "A-OK"';
```

You may choose to use single quotation marks or double quotation marks based on any single or double quotation marks that exist in a string literal, as in the following:

```
var str1:String = "ActionScript <span class='heavy'>3.0</span>";
var str2:String = '<item id="155">banana</item>';
```

Keep in mind that ActionScript distinguishes between a straight single quotation mark (`'`) and a left or right single quotation mark (‘ or ’). The same is true for double quotation marks. Use straight quotation marks to delineate string literals. When pasting text from another source into ActionScript, be sure to use the correct characters.

As the following table shows, you can use the backslash escape character (`\`) to define other characters in string literals:

Escape sequence	Character
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\u`*nnnn*	The Unicode character with the character code specified by the hexadecimal number *nnnn*; for example, `\u263a` is the smiley character.
`\x`*nn*	The ASCII character with the character code specified by the hexadecimal number *nn*
`\'`	Single quotation mark
`\"`	Double quotation mark
`\\`	Single backslash character

The length property

Every string has a `length` property, which is equal to the number of characters in the string:

```
var str:String = "macromedia";
trace(str.length);           // 10
```

An empty string and a null string both have a length of 0, as the following example shows:

```
var str1:String = new String();
trace(str1.length);          // 0

str2:String = '';
trace(str2.length);          // 0
```

Working with characters in strings

Every character in a string has an index position in the string (an integer). The index position of the first character is 0. For example, in the following string, the character `y` is in position 0 and the character `w` is in position 5:

```
"yellow"
```

You can examine individual characters in various positions in a string using the `charAt()` method and the `charCodeAt()` method, as in this example:

```
var str:String = "hello world!";
for (var:i = 0; i < str.length; i++)
{
  trace(str.charAt(i) + " - " + str.charCodeAt(i));
}
```

When you run this code, the following output is produced:

```
h - 104
e - 101
l - 108
l - 108
o - 111
  - 32
w - 119
o - 111
r - 114
l - 108
d - 100
! - 33
```

You can also use character codes to define a string using the `fromCharCode()` method, as the following example shows:

```
var myStr:String =
  String.fromCharCode(104,101,108,108,111,32,119,111,114,108,100,33);
    // Sets myStr to "hello world!"
```

Comparing strings

You can use the following operators to compare strings: <, <=, !=, ==, =>, and >. These operators can be used with conditional statements, such as `if` and `while`, as the following example shows:

```
var str1:String = "Apple";
var str2:String = "apple";
if (str1 < str2)
{
  trace("A < a, B < b, C < c, ...");
}
```

When using these operators with strings, ActionScript considers the character code value of each character in the string, comparing characters from left to right, as in the following:

```
trace("A" < "B"); // true
trace("A" < "a"); // true
trace("Ab" < "az"); // true
trace("abc" < "abza"); // true
```

Use the == and != operators to compare strings with each other and to compare strings with other types of objects, as the following example shows:

```
var str1:String = "1";
var str1b:String = "1";
var str2:String = "2";
trace(str1==str1b); // true
trace(str1==str2); // false
var total:uint = 1;
trace(str1 == total); // true
```

Use the === and !== operators to verify that both comparison objects are of the same type, with the same content:

```
var str1:String = "4";
var str2:String = "4";
var total:uint = 4;
trace(str1 === str2); // true
trace(str1 === total); // false
```

Obtaining string representations of other objects

You can obtain a String representation for any kind of object. All objects have a `toString()` method for this purpose:

```
var n:Number = 99.47;
var str:String = n.toString();
  // str == "99.47"
```

When using the + concatenation operator with a combination of String objects and objects that are not strings, you do not need to use the `toString()` method. For details on concatenation, see the next section.

The `String()` global function returns the same value for a given object as the value returned by the object calling the `toString()` method.

Concatenating strings

Concatenation of strings means taking two strings and joining them sequentially into one. For example, you can use the + operator to concatenate two strings:

```
var str1:String = "green";
var str2:String = "ish";
var str3:String = str1 + str2;
     // str3 == "greenish"
```

You can also use the += operator to the produce the same result, as the following example shows:

```
var str:String = "green";
str += "ish";
     // str == "greenish"
```

Additionally, the String class includes a `concat()` method, as follows:

```
var str1:String = "Bonjour";
var str2:String = "from";
var str3:String = "Paris";
var str4:String = concat(str1, " ", str2, " ", str3)
     // str4 == "Bonjour from Paris"
```

If you use the + operator (or the += operator) with a String object and an object that is *not* a string, ActionScript automatically converts the nonstring object to a String object in order to evaluate the expression, as shown in this example:

```
var str:String = "Area = ";
var area:Number = Math.PI * Math.pow(3, 2);
```

```
str = str + area;
       // str == "Area = 28.274333882308138"
```

However, you can use parentheses for grouping to provide context for the + operator, as the following example shows:

```
trace("Total: $" + 4.55 + 1.45);
     //"Total: $4.551.45"
trace("Total: $" + (4.55 + 1.45));
     //"Total: $6"
```

Finding substrings and patterns in strings

Substrings are sequential characters within a string. For example, the string `"abc"` has the following substrings: `""`, `"a"`, `"ab"`, `"abc"`, `"b"`, `"bc"`, `"c"`. You can use ActionScript methods to locate substrings of a string.

Patterns are defined in ActionScript by strings or by regular expressions. For example, the following regular expression defines a specific pattern—the letters A, B, and C followed by a digit character (the forward slashes are regular expression delimiters):

```
/ABC\d/
```

ActionScript includes methods for finding patterns in strings and for replacing found matches with replacement substrings. These methods are described in the following sections.

Regular expressions can define intricate patterns. For more information, see Chapter 10, "Using Regular Expressions," on page 285.

Finding a substring by character position

The `substr()` and `substring()` methods are similar. Both return a substring of a string. Both take two parameters. In both methods, the first parameter is the position of the starting character in the given string. However, in the `substr()` method, the second parameter is the *length* of the substring to return, and in the `substring()` method, the second parameter is the position of the character at the *end* of the substring (which is not included in the returned string). This example shows the difference between these two methods:

```
var str:String = "Hello from Paris, Texas!!!";
trace(str.substr(11,15)); // Paris, Texas!!!
trace(str.substring(11,15)); // output: Pari
```

The `slice()` method functions similarly to the `substring()` method. When given two non-negative integers as parameters, it works exactly the same. However, the `slice()` method can take negative integers as parameters, in which case the character position is taken from the end of the string, as shown in the following example:

```
var str:String = "Hello from Paris, Texas!!!";
trace(str.slice(11,15)); // Pari
trace(str.slice(-3,-1)); // !!
trace(str.slice(-3,26)); // !!!
trace(str.slice(-3,str.length)); // !!!
trace(str.slice(-8,-3)); // Texas
```

You can combine non-negative and negative integers as the parameters of the `slice()` method.

Finding the character position of a matching substring

You can use the `indexOf()` and `lastIndexOf()` methods to locate matching substrings within a string, as the following example shows:

```
var str:String = "The moon, the stars, the sea, the land";
trace(str.indexOf("the")); // 10
```

Notice that the `indexOf()` method is case-sensitive.

You can specify a second parameter to indicate the index position in the string from which to start the search, as follows:

```
var str:String = "The moon, the stars, the sea, the land"
trace(str.indexOf("the", 11)); // 21
```

The `lastIndexOf()` method finds the last occurrence of a substring in the string:

```
var str:String = "The moon, the stars, the sea, the land"
trace(str.lastIndexOf("the")); // 30
```

If you include a second parameter with the `lastIndexOf()` method, the search is conducted from that index position in the string working backward (from right to left):

```
var str:String = "The moon, the stars, the sea, the land"
trace(str.lastIndexOf("the", 29)); // 21
```

Creating an array of substrings segmented by a delimiter

You can use the `split()` method to create an array of substrings, which is divided based on a delimiter. For example, you can segment a comma-delimited or tab-delimited string into multiple strings.

The following example shows how to split an array into substrings with the ampersand (&) character as the delimiter:

```
var queryStr:String = "first=joe&last=cheng&title=manager&StartDate=3/6/
  65";
var params:Array = queryStr.split("&", 2);
    //     params == ["first=joe","last=cheng"]
```

The second parameter of the `split()` method, which is optional, defines the maximum size of the array that is returned.

You can also use a regular expression as the delimiter character:

```
var str:String = "Give me\t5."
var a:Array = str.split(/\s+/);
  // a == ["Give","me","5."]
```

For more information, see Chapter 10, "Using Regular Expressions," on page 285 and the *ActionScript 3.0 Language Reference*.

Finding patterns in strings and replacing substrings

The String class includes the following methods for working with patterns in strings:

- Use the `match()` and `search()` methods to locate substrings that match a pattern.
- Use the `replace()` method to find substrings that match a pattern and replace them with a specified substring.

These methods are described in the following sections.

You can use strings or regular expressions to define patterns used in these methods. For more information on regular expressions, see Chapter 10, "Using Regular Expressions," on page 285.

Finding matching substrings

The `search()` method returns the index position of the first substring that matches a given pattern, as shown in this example:

```
var str:String = "The more the merrier.";
trace(str.search("the"));
  // output: 9
  // (This search is case-sensitive.)
```

You can also use regular expressions to define the pattern to match, as this example shows:

```
var pattern:RegExp = /the/i;
var str:String = "The more the merrier.";
trace(str.search(pattern)); // 0
```

The output of the `trace()` method is 0, because the first character in the string is index position 0. The `i` flag is set in the regular expression, so the search is not case-sensitive.

The `search()` method finds only one match and returns its starting index position, even if the `g` (global) flag is set in the regular expression.

The following example shows a more intricate regular expression, one that matches a string in double quotation marks:

```
var pattern:RegExp = /"[^"]*"/;
var str:String = "The \"more\" the merrier.";
trace(str.search(pattern));
  // output: 4

str = "The \"more the merrier.";
trace(str.search(pattern));
  // output: -1
  // (Indicates no match, since there is no closing double quotation mark.)
```

The `match()` method works similarly. It searches for a matching substring. However, when you use the global flag in a regular expression pattern, as in the following example, `match()` returns an array of matching substrings:

```
var str:String = "bob@example.com, omar@example.org";
var pattern:RegExp = /\w*@\w*\.[org|com]+/g;
var results:Array = str.match(pattern);
```

The `results` array is set to the following:

```
["bob@example.com","omar@example.org"]
```

For more information on regular expressions, see Chapter 10, “Using Regular Expressions,” on page 285.

Replacing matched substrings

You can use the `replace()` method to search for a specified pattern in a string and replace matches with the specified replacement string, as the following example shows:

```
var str:String = "She sells seashells by the seashore.";
var pattern:RegExp = /sh/gi;
trace(str.replace(pattern, "sch"));
  //sche sells seaschells by the seaschore.
```

Note that in this example, the matched strings are not case-sensitive because the `i` (`ignoreCase`) flag is set in the regular expression, and multiple matches are replaced because the `g` (`global`) flag is set. For more information, see Chapter 10, "Using Regular Expressions," on page 285.

You can include the following `$` replacement codes in the replacement string. The replacement text shown in the following table is inserted in place of the `$` replacement code:

$ Code	Replacement Text
`$$`	$
`$&`	The matched substring.
`` $` ``	The portion of the string that precedes the matched substring. This code uses the straight left single quotation mark character (`` ` ``), not the straight single quotation mark (`'`) or the left curly single quotation mark (‘).
`$'`	The portion of the string that follows the matched substring. This code uses the straight single quotation mark (’).
`$`*n*	The *n*th captured parenthetical group match, where *n* is a single digit, 1-9, and **$***n* is not followed by a decimal digit.
`$`*nn*	The *nn*th captured parenthetical group match, where *nn* is a two-digit decimal number, 01–99. If the *nn*th capture is undefined, the replacement text is an empty string.

For example, the following shows the use of the `$2` and `$1` replacement codes, which represent the first and second capturing group matched:

```
var str:String = "flip-flop";
var pattern:RegExp = /(\w+)-(\w+)/g;
trace(str.replace(pattern, "$2-$1")); // flop-flip
```

You can also use a function as the second parameter of the `replace()` method. The matching text is replaced by the returned value of the function.

```
var str:String = "Now only $9.95!";
var price:RegExp = /\$([\d,]+.\d+)+/i;
trace(str.replace(price, usdToEuro));
```

```
function usdToEuro(matchedSubstring:String,
                   capturedMatch1:String,
                   index:int,
                   str:String):String
{
  var usd:String = capturedMatch1;
  usd = usd.replace(",", "");
  var exchangeRate:Number = 0.853690;
  var euro:Number = usd * exchangeRate;
  const euroSymbol:String = String.fromCharCode(8364);
  return euro.toFixed(2) + " " + euroSymbol;
}
```

When you use a function as the second parameter of the `replace()` method, the following arguments are passed to the function:

- The matching portion of the string.
- Any capturing parenthetical group matches. The number of arguments passed this way will vary depending on the number of parenthetical matches. You can determine the number of parenthetical matches by checking `arguments.length - 3` within the function code.
- The index position in the string where the match begins.
- The complete string.

Converting strings between uppercase and lowercase

As the following example shows, the `toLowerCase()` method and the `toUpperCase()` method convert alphabetical characters in the string to lowercase and uppercase, respectively:

```
var str:String = "Dr. Bob Roberts, #9."
trace(str.toLowerCase());
  // dr. bob roberts, #9.
trace(str.toUpperCase());
  // DR. BOB ROBERTS, #9.
```

After these methods are executed, the source string remains unchanged. To transform the source string, use the following code:

```
str = str.toUpperCase();
```

These methods work with extended characters, not simply a–z and A–Z:

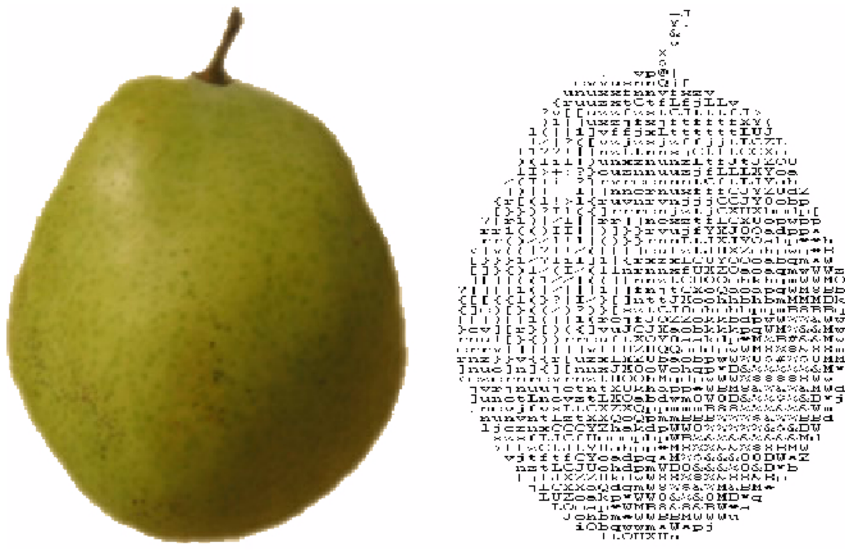
```
var str:String = "José Barça";
trace(str.toUpperCase(), str.toLowerCase()); // JOSÉ BARÇA josé barça
```

Example: ASCII Art

This ASCII Art example shows a number of features of working with the String class in ActionScript 3.0, including the following:

- The `split()` method of the String class is used to extract values from a character-delimited string (image information in a tab-delimited text file).
- Several string-manipulation techniques, including `split()`, concatenation, and extracting a portion of the string using `substring()` and `substr()`, are used to capitalize the first letter of each word in the image titles.
- The `getCharAt()` method is used to get a single character from a string (to determine the ASCII character corresponding to a grayscale bitmap value).
- String concatenation is used to build up the ASCII art representation of an image one character at a time.

The term *ASCII art* refers to a text representations of an image, in which a grid of monospaced font characters, such as Courier New characters, plots the image. The following image shows an example of ASCII art produced by the application:



The ASCII art version of the graphic is shown on the right.

The ASCIIArt application files can be found in the folder Samples/AsciiArt. The application consists of the following files:

File	Description
AsciiArtApp.mxml	The user interface for the application in MXML for Flex.
com/example/programmingas3/asciiArt/ AsciiArtBuilder.as	The class that provides the main functionality of the application, including extracting image metadata from a text file, loading the images, and managing the image-to-text conversion process.
com/example/programmingas3/asciiArt/ BitmapToAsciiConverter.as	A class that provides the `parseBitmapData()` method for converting image data into a String version.
com/example/programmingas3/asciiArt/ Image.as	A class which represents a loaded bitmap image.
com/example/programmingas3/asciiArt/ ImageInfo.as	A class representing metadata for an ASCII art image (such as title, image file URL, and so on).
image/	A folder containing images used by the application.
txt/ImageData.txt	A tab-delimited text file, containing information on the images to be loaded by the application.

Extracting tab-delimited values

This example uses the common practice of storing application data separate from the application itself; that way, if the data changes (for example, if another image is added or an image's title changes), there is no need to recreate the SWF file. In this case, the image metadata, including the image title, the URL of the actual image file, and some values that are used to manipulate the image, are stored in a text file (the txt/ImageData.txt file in the project). The contents of the text file are as follows:

```
FILENAMETITLEWHITE_THRESHHOLDBLACK_THRESHHOLD
FruitBasket.jpgPear, apple, orange, and bananad810
Banana.jpgA picture of a bananaC820
Orange.jpgorangeFF20
Apple.jpgpicture of an apple6E10
```

The file uses a specific tab-delimited format. The first line (row) is a heading row. The remaining lines contain the following data for each bitmap to be loaded:

- The filename of the bitmap.
- The display name of the bitmap.
- The white-threshold and black-threshold values for the bitmaps. These are hex values above which and below which a pixel is to be considered completely white or completely black.

As soon as the application starts, the AsciiArtBuilder class loads and parses the contents of the text file in order to create the “stack” of images that it will display, using the following code from the AsciiArtBuilder class’s `parseImageInfo()` method:

```
var lines:Array = _imageInfoLoader.data.split("\n");
var numLines:uint = lines.length;
for (var i:uint = 1; i < numLines; i++)
{
  var imageInfoRaw:String = lines[i];
  ...
  if (imageInfoRaw.length > 0)
  {
    // Create a new image info record and add it to the array of image info.
    var imageInfo:ImageInfo = new ImageInfo();

    // Split the current line into values (separated by tab (\t)
    // characters) and extract the individual properties:
    var imageProperties:Array = imageInfoRaw.split("\t");
    imageInfo.fileName = imageProperties[0];
    imageInfo.title = normalizeTitle(imageProperties[1]);
    imageInfo.whiteThreshold = parseInt(imageProperties[2], 16);
    imageInfo.blackThreshold = parseInt(imageProperties[3], 16);
    result.push(imageInfo);
  }
}
```

The entire contents of the text file are contained in a single String instance, the `_imageInfoLoader.data` property. Using the `split()` method with the newline character (`"\n"`) as a parameter, the String instance is divided into an Array (`lines`) whose elements are the individual lines of the text file. Next, the code uses a loop to work with each of the lines (except the first, because it contains only headers rather than actual content). Inside the loop, the `split()` method is used once again to divide the contents of the single line into a set of values (the Array object named `imageProperties`). The parameter used with the `split()` method in this case is the tab (`"\t"`) character, because the values in each line are delineated by tab characters.

Using String methods to normalize image titles

One of the design decisions for this application is that all the image titles are displayed using a standard format, with the first letter of each word capitalized (except for a few words which are commonly not capitalized in English titles). Rather than assume that the text file contains properly formatted titles, the application formats the titles while they're being extracted from the text file.

In the previous code listing, as part of extracting individual image metadata values, the following line of code is used:

```
imageInfo.title = normalizeTitle(imageProperties[1]);
```

In that code, the image's title from the text file is passed through the `normalizeTitle()` method before it is stored in the ImageInfo object:

```
private function normalizeTitle(title:String):String
{
  var words:Array = title.split(" ");
  var len:uint = words.length;
  for(var i:uint; i < len; i++)
  {
    words[i] = capitalizeFirstLetter(words[i]);
  }

  return words.join(" ");
}
```

This method uses the `split()` method to divide the title into individual words (separated by the space character), passes each word through the `capitalizeFirstLetter()` method, and then uses the Array class's `join()` method to combine the words back into a single string again.

As its name suggests, the `capitalizeFirstLetter()` method actually does the work of capitalizing the first letter of each word:

```
/**
 * Capitalizes the first letter of a single word, unless it's one of
 * a set of words that are normally not capitalized in English.
 */
private function capitalizeFirstLetter(word:String):String
{
  switch (word)
  {
    case "and":
    case "the":
    case "in":
    case "an":
    case "or":
    case "at":
    case "of":
    case "a":
      // Don't do anything to these words.
      break;
    default:
      // For any other word, capitalize the first character.
      var firstLetter:String = word.substr(0, 1);
      firstLetter = firstLetter.toUpperCase();
      var otherLetters:String = word.substring(1);
      word = firstLetter + otherLetters;
  }
  return word;
}
```

In English, the initial character of each word in a title is *not* capitalized if it is one of the following words: "and," "the," "in," "an," "or," "at," "of," or "a." (This is a simplified version of the rules.) To execute this logic, the code first uses a `switch` statement to check if the word is one of the words that should not be capitalized. If so, the code simply jumps out of the `switch` statement. On the other hand, if the word should be capitalized, that is done in several steps, as follows:

1. The first letter of the word is extracted using `substr(0, 1)`, which extracts a substring starting with the character at index 0 (the first letter in the string, as indicated by the first parameter `0`). The substring will be one character in length (indicated by the second parameter `1`).
2. That character is capitalized using the `toUpperCase()` method.

3. The remaining characters of the original word are extracted using `substring(1)`, which extracts a substring starting at index 1 (the second letter) through the end of the string (indicated by leaving off the second parameter of the `substring()` method).
4. The final word is created by combining the newly capitalized first letter with the remaining letters using string concatenation: `firstLetter + otherLetters`.

Generating the ASCII art text

The BitmapToAsciiConverter class provides the functionality of converting a bitmap image to its ASCII text representation. This process is performed by the `parseBitmapData()` method, which is partially shown here:

```
var result:String = "";

// Loop through the rows of pixels top to bottom:
for (var y:uint = 0; y < _data.height; y += verticalResolution)
{
    // Within each row, loop through pixels left to right:
    for (var x:uint = 0; x < _data.width; x += horizontalResolution)
    {
        ...

        // Convert the gray value in the 0-255 range to a value
        // in the 0-64 range (since that's the number of "shades of
        // gray" in the set of available characters):
        index = Math.floor(grayVal / 4);
        result += palette.charAt(index);
    }
    result += "\n";
}
return result;
```

This code first defines a String instance named `result` that will be used to build up the ASCII art version of the bitmap image. Next, it loops through individual pixels of the source bitmap image. Using several color-manipulation techniques (omitted here for brevity), it converts the red, green, and blue color values of an individual pixel to a single grayscale value (a number from 0 to 255). The code then divides that value by 4 (as shown) to convert it to a value in the 0-63 scale, which is stored in the variable `index`. (The 0-63 scale is used because the "palette" of available ASCII characters used by this application contains 64 values.) The palette of characters is defined as a String instance in the BitmapToAsciiConverter class:

```
// The characters are in order from darkest to lightest, so that their
// position (index) in the string corresponds to a relative color value
// (0 = black).
private static const palette:String =
  "@#$%&8BMW*mwqpdbkhaoQ0OZXYUJCLtfjzxnuvcr[]{}1()|/?Il!i><+_~-;,. ";
```

Since the `index` variable defines which ASCII character in the palette corresponds to the current pixel in the bitmap image, that character is retrieved from the `palette` String using the `charAt()` method. It is then appended to the `result` String instance using the concatenation assignment operator (+=). In addition, at the end of each row of pixels, a newline character is concatenated to the end of the `result` String, forcing the line to wrap to create a new row of character "pixels."

CHAPTER 8

Working with Arrays

8

Arrays allow you to store multiple values in a single data structure. You can use simple indexed arrays that store values using fixed ordinal integer indexes or complex associative arrays that store values using arbitrary keys. Arrays can also be multidimensional, containing elements that are themselves arrays. This chapter discusses how to create and manipulate various types of arrays.

Contents

Indexed arrays

Indexed arrays store a series of one or more values organized such that each value can be accessed using an unsigned integer value. The first index is always the number 0, and the index increments by 1 for each subsequent element added to the array. As the following code shows, you can create an indexed array by calling the Array class constructor or by initializing the array with an array literal:

```
// Use Array constructor.
var myArray:Array = new Array();
myArray.push("one");
myArray.push("two");
myArray.push("three");
trace(myArray); // output: one,two,three

// Use Array literal.
```

```
var myArray:Array = ["one", "two", "three"];
trace(myArray); // output: one,two,three
```

The Array class also contains properties and methods that allow you to modify indexed arrays. These properties and methods apply almost exclusively to indexed arrays rather than associative arrays.

Indexed arrays use an unsigned 32-bit integer for the index number. The maximum size of an indexed array is 2^{32}-1 or 4,294,967,295. An attempt to create an array that is larger than the maximum size results in a run-time error.

An array element can hold a value of any data type. ActionScript 3.0 does not support the concept of *typed arrays*, which means that you cannot specify that all the elements of an array belong to a specific data type.

This section explains how to create and modify indexed arrays using the Array class, starting with how to create an array. The methods that modify arrays are grouped into three categories that cover how to insert elements, remove elements, and sort arrays. Methods in a final group treat an existing array as read-only; these methods merely query arrays. Rather than modifying an existing array, the query methods all return a new array. The section concludes with a discussion about extending the Array class.

Creating arrays

The Array constructor function can be used in three ways. First, if you call the constructor with no arguments, you get an empty array. You can use the `length` property of the Array class to verify that the array has no elements. For example, the following code calls the Array constructor with no arguments:

```
var names:Array = new Array();
trace(names.length); // output: 0
```

Second, if you use a number as the only parameter to the Array constructor, an array of that length is created, with each element's value set to `undefined`. The argument must be an unsigned integer between the values 0 and 4,294,967,295. For example, the following code calls the Array constructor with a single numeric argument:

```
var names:Array = new Array(3);
trace(names.length); // output: 3
trace(names[0]); // output: undefined
trace(names[1]); // output: undefined
trace(names[2]); // output: undefined
```

Third, if you call the constructor and pass a list of elements as parameters, an array is created with elements corresponding to each of the parameters. The following code passes three arguments to the Array constructor:

```
var names:Array = new Array("John", "Jane", "David");
trace(names.length); // output: 3
trace(names[0]); // output: John
trace(names[1]); // output: Jane
trace(names[2]); // output: David
```

You can also create arrays with array literals or object literals. An array literal can be assigned directly to an array variable, as shown in the following example:

```
var names:Array = ["John", "Jane", "David"];
```

Inserting array elements

Three of the Array class methods—`push()`, `unshift()`, and `splice()`—allow you to insert elements into an array. The `push()` method appends one or more elements to the end of an array. In other words, the last element inserted into the array using the `push()` method will have the highest index number. The `unshift()` method inserts one or more elements at the beginning of an array, which is always at index number 0. The `splice()` method will insert any number of items at a specified index in the array.

The following example demonstrates all three methods. An array named `planets` is created to store the names of the planets in order of proximity to the Sun. First, the `push()` method is called to add the initial item, `Mars`. Second, the `unshift()` method is called to insert the item that belongs at the front of the array, `Mercury`. Finally, the `splice()` method is called to insert the items `Venus` and `Earth` after `Mercury`, but before `Mars`. The first argument sent to `splice()`, the integer 1, directs the insertion to begin at index 1. The second argument sent to `splice()`, the integer 0, indicates that no items should be deleted. Finally, the third and fourth arguments sent to `splice()`, `Venus` and `Earth`, are the items to be inserted.

```
var planets:Array = new Array();
planets.push("Mars");       // array contents: Mars
planets.unshift("Mercury"); // array contents: Mercury,Mars
planets.splice(1, 0, "Venus", "Earth");
trace(planets);             // array contents: Mercury,Venus,Earth,Mars
```

The `push()` and `unshift()` methods both return an unsigned integer that represents the length of the modified array. The `splice()` method returns an empty array when used to insert elements, which may seem strange, but makes more sense in light of the `splice()` method's versatility. You can use the `splice()` method not only to insert elements into an array, but also to remove elements from an array. When used to remove elements, the `splice()` method returns an array containing the elements removed.

Removing array elements

Three methods of the Array class—`pop()`, `shift()`, and `splice()`—allow you to remove elements from an array. The `pop()` method removes an element from the end of the array. In other words, it removes the element at the highest index number. The `shift()` method removes an element from the beginning of the array, which means that it always removes the element at index number 0. The `splice()` method, which can also be used to insert elements, removes an arbitrary number of elements starting at the index number specified by the first argument sent to the method.

The following example uses all three methods to remove elements from an array. An array named `oceans` is created to store the names of large bodies of water. Some of the names in the array are lakes rather than oceans, so they need to be removed.

First, the `splice()` method is used to remove the items `Aral` and `Superior`, and insert the items `Atlantic` and `Indian`. The first argument sent to `splice()`, the integer 2, indicates that the operation should start with the third item in the list, which is at index 2. The second argument, 2, indicates that two items should be removed. The remaining arguments, `Atlantic` and `Indian`, are values to be inserted at index 2.

Second, the `pop()` method is used to remove last element in the array, `Huron`. And third, the `shift()` method is used to remove the first item in the array, `Victoria`.

```
var oceans:Array = ["Victoria", "Pacific", "Aral", "Superior", "Indian",
  "Huron"];
oceans.splice(2, 2, "Arctic", "Atlantic"); // replaces Aral and Superior
oceans.pop();   // removes Huron
oceans.shift(); // removes Victoria
trace(oceans);  // output: Pacific,Arctic,Atlantic,Indian
```

The `pop()` and `shift()` methods both return the item that was removed. The data type of the return value is Object because arrays can hold values of any data type. The `splice()` method returns an array containing the values removed. You can change the oceans array example so that the call to `splice()` assigns the array to a new array variable, as shown in the following example:

```
var lakes:Array = oceans.splice(2, 2, "Arctic", "Atlantic");
trace(lakes); // output: Aral,Superior
```

You may come across code that uses the `delete` operator on an array element. The `delete` operator sets the value of an array element to `undefined`, but it does not remove the element from the array. For example, the following code uses the `delete` operator on the third element in the `oceans` array, but the length of the array remains 5:

```
var oceans:Array = ["Arctic", "Pacific", "Victoria", "Indian", "Atlantic"];
delete oceans[2];
trace(oceans);       // output: Arctic,Pacific,,Indian,Atlantic
trace(oceans[2]);      // output: undefined
trace(oceans.length); // output: 5
```

You can truncate an array using an array's `length` property. If you set the `length` property of an array to a length that is less than the current length of the array, the array is truncated, removing any elements stored at index numbers higher than the new value of `length` minus 1. For example, if the `oceans` array were sorted such that all valid entries were at the beginning of the array, you could use the `length` property to remove the entries at the end of the array, as shown in the following code:

```
var oceans:Array = ["Arctic", "Pacific", "Victoria", "Aral", "Superior"];
oceans.length = 2;
trace(oceans); // output: Arctic,Pacific
```

Sorting an array

There are three methods—`reverse()`, `sort()`, and `sortOn()`—that allow you to change the order of an array, either by sorting or reversing the order. All of these methods modify the existing array. The `reverse()` method changes the order of the array such that the last element becomes the first element, the penultimate element becomes the second element, and so on. The `sort()` method allows you to sort an array in a variety of predefined ways, and even allows you to create custom sorting algorithms. The `sortOn()` method allows you to sort an indexed array of objects that have one or more common properties that can be used as sort keys.

The `reverse()` method takes no parameters and does not return a value, but allows you to toggle the order of your array from its current state to the reverse order. The following example reverses the order of the oceans listed in the `oceans` array:

```
var oceans:Array = ["Arctic", "Atlantic", "Indian", "Pacific"];
oceans.reverse();
trace(oceans); // output: Pacific,Indian,Atlantic,Arctic
```

The `sort()` method rearranges the elements in an array using the *default sort order*. The default sort order has the following characteristics:

- The sort is case-sensitive, which means that uppercase characters precede lowercase characters. For example, the letter D precedes the letter b.
- The sort is ascending, which means that lower character codes (such as A) precede higher character codes (such as B).
- The sort places identical values adjacent to each other but in no particular order.
- The sort is string-based, which means that elements are converted to strings before they are compared (for example, 10 precedes 3 because the string `"1"` has a lower character code than the string `"3"` has).

You may find that you need to sort your array without regard to case, or in descending order, or perhaps your array contains numbers that you want to sort numerically instead of alphabetically. The `sort()` method has an `options` parameter that allows you to alter each characteristic of the default sort order. The options are defined by a set of static constants in the Array class, as shown in the following list:

- `Array.CASEINSENSITIVE`: This option makes the sort disregard case. For example, the lowercase letter b precedes the uppercase letter D.
- `Array.DESCENDING`: This reverses the default ascending sort. For example, the letter B precedes the letter A.
- `Array.UNIQUESORT`: This causes the sort to abort if two identical values are found.
- `Array.NUMERIC`: This causes numerical sorting, so that 3 precedes 10.

The following example highlights some of these options. An array named `poets` is created that is sorted using several different options.

```
var poets:Array = ["Blake", "cummings", "Angelou", "Dante"];
poets.sort(); // default sort
trace(poets); // output: Angelou,Blake,Dante,cummings

poets.sort(Array.CASEINSENSITIVE);
trace(poets); // output: Angelou,Blake,cummings,Dante

poets.sort(Array.DESCENDING);
trace(poets); // output: cummings,Dante,Blake,Angelou

poets.sort(Array.DESCENDING | Array.CASEINSENSITIVE); // use two options
trace(poets); // output: Dante,cummings,Blake,Angelou
```

You can also write your own custom sort function, which you can pass as a parameter to the `sort()` method. For example, if you have a list of names in which each list element contains a person's full name, but you want to sort the list by last name, you must use a custom sort function to parse each element and use the last name in the sort function. The following code shows how this can be done with a custom function that is used as a parameter to the `Array.sort()` method:

```
var names:Array = new Array("John Q. Smith", "Jane Doe", "Mike Jones");
function orderLastName(a, b):int
{
  var lastName:RegExp = /\b\S+$/;
  var name1 = a.match(lastName);
  var name2 = b.match(lastName);
  if (name1 < name2)
  {
    return -1;
  }
  else if (name1 > name2)
```

```
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
trace(names); // output: John Q. Smith,Jane Doe,Mike Jones
names.sort(orderLastName);
trace(names); // output: Jane Doe,Mike Jones,John Q. Smith
```

The custom sort function `orderLastName()` uses a regular expression to extract the last name from each element to use for the comparison operation. The function identifier `orderLastName` is used as the sole parameter when calling the `sort()` method on the `names` array. The sort function accepts two parameters, `a` and `b`, because it works on two array elements at a time. The sort function's return value indicates how the elements should be sorted:

- A return value of -1 indicates that the first parameter, `a`, precedes the second parameter, `b`.
- A return value of 1 indicates that the second parameter, `b`, precedes the first, `a`.
- A return value of 0 indicates that the elements have equal sorting precedence.

The `sortOn()` method is designed for indexed arrays with elements that contain objects. These objects are expected to have at least one common property that can be used as the sort key. The use of the `sortOn()` method for arrays of any other type yields unexpected results.

The following example revises the `poets` array so that each element is an object instead of a string. Each object holds both the poet's last name and year of birth.

```
var poets:Array = new Array();
poets.push({name:"Angelou", born:"1928"});
poets.push({name:"Blake", born:"1757"});
poets.push({name:"cummings", born:"1894"});
poets.push({name:"Dante", born:"1265"});
poets.push({name:"Wang", born:"701"});
```

You can use the `sortOn()` method to sort the array by the `born` property. The `sortOn()` method defines two parameters, `fieldName` and `options`. The `fieldName` argument must be specified as a string. In the following example, `sortOn()` is called with two arguments, `"born"` and `Array.NUMERIC`. The `Array.NUMERIC` argument is used to ensure that the sort is done numerically instead of alphabetically. This is a good practice even when all the numbers have the same number of digits because it ensures that the sort will continue to behave as expected if a number with fewer or more digits is later added to the array.

```
poets.sortOn("born", Array.NUMERIC);
for (var i:int = 0; i < poets.length; ++i)
```

```
{
  trace(poets[i].name, poets[i].born);
}
/* output:
Wang 701
Dante 1265
Blake 1757
cummings 1894
Angelou 1928
*/
```

Generally, the `sort()` and `sortOn()` methods modify an array. If you wish to sort an array without modifying the existing array, pass the `Array.RETURNINDEXEDARRAY` constant as part of the `options` parameter. This option directs the methods to return a new array that reflects the sort and to leave the original array unmodified. The array returned by the methods is a simple array of index numbers that reflects the new sort order and does not contain any elements from the original array. For example, to sort the `poets` array by birth year without modifying the array, include the `Array.RETURNINDEXEDARRAY` constant as part of the argument passed for the `options` parameter.

The following example stores the returned index information in an array named `indices` and uses the `indices` array in conjunction with the unmodified `poets` array to output the poets in order of birth year:

```
var indices:Array;
indices = poets.sortOn("born", Array.NUMERIC | Array.RETURNINDEXEDARRAY);
for (var i:int = 0; i < indices.length; ++i)
{
  var index:int = indices[i];
  trace(poets[index].name, poets[index].born);
}
/* output:
Wang 701
Dante 1265
Blake 1757
cummings 1894
Angelou 1928
*/
```

Querying an array

The remaining four methods of the Array class—`concat()`, `join()`, `slice()`, `toString()`—all query the array for information, but do not modify the array. The `concat()` and `slice()` methods both return new arrays, while the `join()` and `toString()` methods both return strings. The `concat()` method takes a new array or list of elements as arguments and combines it with the existing array to create a new array. The `slice()` method has two parameters, aptly named `startIndex` and an `endIndex`, and returns a new array containing a copy of the elements "sliced" from the existing array. The slice begins with the element at `startIndex` and ends with the element just before `endIndex`. That bears repeating: the element at `endIndex` is not included in the return value.

The following example uses `concat()` and `slice()` to create new arrays using elements of other arrays:

```
var array1:Array = ["alpha", "beta"];
var array2:Array = array1.concat("gamma", "delta");
trace(array2); // output: alpha,beta,gamma,delta

var array3:Array = array1.concat(array2);
trace(array3); // output: alpha,beta,alpha,beta,gamma,delta

var array4:Array = array3.slice(2,5);
trace(array4); // output: alpha,beta,gamma
```

You can use the `join()` and `toString()` methods to query the array and return its contents as a string. If no parameters are used for the `join()` method, the two methods behave identically—they return a string containing a comma-delimited list of all elements in the array. The `join()` method, unlike the `toString()` method, accepts a parameter named `delimiter`, which allows you to choose the symbol to use as a separator between each element in the returned string.

The following example creates an array called `rivers` and calls both `join()` and `toString()` to return the values in the array as a string. The `toString()` method is used to return comma-separated values (`riverCSV`), while the `join()` method is used to return values separated by the + character.

```
var rivers:Array = ["Nile", "Amazon", "Yangtze", "Mississippi"];
var riverCSV:String = rivers.toString();
trace(riverCSV); // output: Nile,Amazon,Yangtze,Mississippi
var riverPSV:String = rivers.join("+");
trace(riverPSV); // output: Nile+Amazon+Yangtze+Mississippi
```

One issue to be aware of with the `join()` method is that any nested arrays are always returned with comma-separated values, no matter what separator you specify for the main array elements, as the following example shows:

```
var nested:Array = ["b","c","d"];
var letters:Array = ["a",nested,"e"];
var joined:String = letters.join("+");
trace(joined); // output: a+b,c,d+e
```

Associative arrays

Associative arrays, which are sometimes called *hashes* or *maps*, use *keys* instead of a numeric index to organize stored values. Each key in an associative array is a unique string that is used to access a stored value. An associative array is an instance of the Object class, which means that each key corresponds to a property name. Associative arrays are unordered collections of key and value pairs. Your code should not expect the keys of an associative array to be in a specific order.

ActionScript 3.0 introduces an advanced type of associative array called a *dictionary.* Dictionaries, which are instances of the Dictionary class in the flash.utils package, use keys that can be of any data type but are usually instances of the Object class. In other words, dictionary keys are not limited to values of type String.

This section describes how to create associative arrays that use strings for keys and how to use the Dictionary class.

Associative arrays with string keys

There are two ways to create associative arrays in ActionScript 3.0. The first way is to use the Object constructor, which has the advantage of allowing you to initialize your array with an object literal. An instance of the Object class, also called a generic object, is functionally identical to an associative array. Each property name of the generic object serves as the key that provides access to a stored value.

The following example creates an associative array named `monitorInfo`, using an object literal to initialize the array with two key and value pairs:

```
var monitorInfo:Object = {type:"Flat Panel", resolution:"1600 x 1200"};
trace (monitorInfo["type"], monitorInfo["resolution"]);
// output: Flat Panel 1600 x 1200
```

If you do not need to initialize the array at declaration time, you can use the Object constructor to create the array, as follows:

```
var monitorInfo:Object = new Object();
```

After the array is created using either an object literal or the Object class constructor, you can add new values to the array using either the bracket operator (`[]`) or the dot operator (`.`). The following example adds two new values to `monitorArray`:

```
monitorInfo["aspect ratio"] = "16:10"; // bad form, do not use spaces
monitorInfo.colors = "16.7 million";
trace (monitorInfo["aspect ratio"], monitorInfo.colors);
// output: 16:10 16.7 million
```

Note that the key named `aspect ratio` contains a space character. This is possible with the bracket operator, but generates an error if attempted with the dot operator. Using spaces in your key names is not recommended.

The second way to create an associative array is to use the Array constructor and then use either the bracket operator (`[]`) or the dot operator (`.`) to add key and value pairs to the array. If you declare your associative array to be of type Array, you cannot use an object literal to initialize the array. The following example creates an associative array named `monitorInfo` using the Array constructor and adds a key called `type` and a key called `resolution`, along with their values:

```
var monitorInfo:Array = new Array();
monitorInfo["type"] = "Flat Panel";
monitorInfo["resolution"] = "1600 x 1200";
trace(monitorInfo["type"], monitorInfo["resolution"]);
// output: Flat Panel 1600 x 1200
```

There is no advantage in using the Array constructor to create an associative array. You cannot use the `Array.length` property or any of the methods of the Array class with associative arrays, even if you use the Array constructor or the Array data type. The use of the Array constructor is best left for the creation of indexed arrays.

Associative arrays with object keys

You can use the Dictionary class to create an associative array that uses objects for keys rather than strings. Such arrays are sometimes called dictionaries, hashes, or maps. For example, consider an application that determines the location of a Sprite object based on its association with a specific container. You can use a Dictionary object to map each Sprite object to a container.

The following code creates three instances of the Sprite class that serve as keys for the Dictionary object. Each key is assigned a value of either `GroupA` or `GroupB`. The values can be of any data type, but in this example both `GroupA` and `GroupB` are instances of the Object class. Subsequently, you can access the value associated with each key with the property access (`[]`) operator, as shown in the following code:

```
import flash.display.Sprite;
import flash.utils.Dictionary;

var groupMap:Dictionary = new Dictionary();

// objects to use as keys
var spr1:Sprite = new Sprite();
var spr2:Sprite = new Sprite();
var spr3:Sprite = new Sprite();

// objects to use as values
var groupA:Object = new Object();
var groupB:Object = new Object();

// Create new key-value pairs in dictionary.
groupMap[spr1] = groupA;
groupMap[spr2] = groupB;
groupMap[spr3] = groupB;

if (groupMap[spr1] == groupA)
{
  trace("spr1 is in groupA");
}
if (groupMap[spr2] == groupB)
{
  trace("spr2 is in groupB");
}
if (groupMap[spr3] == groupB)
{
  trace("spr3 is in groupB");
}
```

Iterating with object keys

You can iterate through the contents of a Dictionary object with either a `for..in` loop or a `for each..in` loop. A `for..in` loop allows you to iterate based on the keys, whereas a `for each..in loop` allows you to iterate based on the values associated with each key.

Use the `for..in` loop for direct access to the object keys of a Dictionary object. You can also access the values of the Dictionary object with the property access (`[]`) operator. The following code uses the previous example of the `groupMap` dictionary to show how to iterate through a Dictionary object with the `for..in` loop:

```
for (var key:Object in groupMap)
{
  trace(key, groupMap[key]);
}
/* output:
[object Sprite] [object Object]
[object Sprite] [object Object]
[object Sprite] [object Object]
*/
```

Use the `for each..in` loop for direct access to the values of a Dictionary object. The following code also uses the `groupMap` dictionary to show how to iterate through a Dictionary object with the `for each..in` loop:

```
for each (var item:Object in groupMap)
{
  trace(item);
}
/* output:
[object Object]
[object Object]
[object Object]
*/
```

Object keys and memory management

Flash Player uses a garbage collection system to recover memory that is no longer used. When an object has no references pointing to it, the object becomes eligible for garbage collection, and the memory is recovered the next time the garbage collection system executes. For example, the following code creates a new object and assigns a reference to the object to the variable `myObject`:

```
var myObject:Object = new Object();
```

As long as any reference to the object exists, the garbage collection system will not recover the memory that the object occupies. If the value of `myObject` is changed such that it points to a different object or is set to the value `null`, the memory occupied by the original object becomes eligible for garbage collection, but only if there are no other references to the original object.

If you use `myObject` as a key in a Dictionary object, you are creating another reference to the original object. For example, the following code creates two references to an object—the `myObject` variable, and the key in the `myMap` object:

```
import flash.utils.Dictionary;

var myObject:Object = new Object();
var myMap:Dictionary = new Dictionary();
myMap[myObject] = "foo";
```

To make the object referenced by `myObject` eligible for garbage collection, you must remove all references to it. In this case, you must change the value of `myObject` and delete the `myObject` key from `myMap`, as shown in the following code:

```
myObject = null;
delete myMap[myObject];
```

Alternatively, you can use the `useWeakReference` parameter of the Dictionary constructor to make all of the dictionary keys *weak references*. The garbage collection system ignores weak references, which means that an object that has only weak references is eligible for garbage collection. For example, in the following code, you do not need to delete the `myObject` key from `myMap` in order to make the object eligible for garbage collection:

```
import flash.utils.Dictionary;

var myObject:Object = new Object();
var myMap:Dictionary = new Dictionary(true);
myMap[myObject] = "foo";
myObject = null; // Make object eligible for garbage collection.
```

Multidimensional arrays

Multidimensional arrays contain other arrays as elements. For example, consider a list of tasks that is stored as an indexed array of strings:

```
var tasks:Array = ["wash dishes", "take out trash"];
```

If you want to store a separate list of tasks for each day of the week, you can create a multidimensional array with one element for each day of the week. Each element contains an indexed array, similar to the `tasks` array, that stores the list of tasks. You can use any combination of indexed or associative arrays in multidimensional arrays. The examples in the following sections use either two indexed arrays or an associative array of indexed arrays. You might want to try the other combinations as exercises.

Two indexed arrays

When you use two indexed arrays, you can visualize the result as a table or spreadsheet. The elements of the first array represent the rows of the table, while the elements of the second array represent the columns.

For example, the following multidimensional array uses two indexed arrays to track task lists for each day of the week. The first array, `masterTaskList`, is created using the Array class constructor. Each element of the array represents a day of the week, with index 0 representing Monday, and index 6 representing Sunday. These elements can be thought of as the rows in the table. You can create each day's task list by assigning an array literal to each of the seven elements that you create in the `masterTaskList` array. The array literals represent the columns in the table.

```
var masterTaskList:Array = new Array();
masterTaskList[0] = ["wash dishes", "take out trash"];
masterTaskList[1] = ["wash dishes", "pay bills"];
masterTaskList[2] = ["wash dishes", "dentist", "wash dog"];
masterTaskList[3] = ["wash dishes"];
masterTaskList[4] = ["wash dishes", "clean house"];
masterTaskList[5] = ["wash dishes", "wash car", "pay rent"];
masterTaskList[6] = ["mow lawn", "fix chair"];
```

You can access individual items on any of the task lists using bracket notation. The first set of brackets represents the day of the week, and the second set of brackets represents the task list for that day. For example, to retrieve the second task from Wednesday's list, first use index 2 for Wednesday, and then use index 1 for the second task in the list.

```
trace(masterTaskList[2][1]); // output: dentist
```

To retrieve the first task from Sunday's list, use index 6 for Sunday and index 0 for the first task on the list.

```
trace(masterTaskList[6][0]); // output: mow lawn
```

Associative array with an indexed array

To make the individual arrays easier to access, you can use an associative array for the days of the week and an indexed array for the task lists. Using an associative array allows you to use dot syntax when referring to a particular day of the week, but at the cost of extra run time processing to access each element of the associative array. The following example uses an associative array as the basis of a task list, with a key and value pair for each day of the week:

```
var masterTaskList:Object = new Object();
masterTaskList["Monday"] = ["wash dishes", "take out trash"];
masterTaskList["Tuesday"] = ["wash dishes", "pay bills"];
masterTaskList["Wednesday"] = ["wash dishes", "dentist", "wash dog"];
masterTaskList["Thursday"] = ["wash dishes"];
masterTaskList["Friday"] = ["wash dishes", "clean house"];
masterTaskList["Saturday"] = ["wash dishes", "wash car", "pay rent"];
masterTaskList["Sunday"] = ["mow lawn", "fix chair"];
```

Dot syntax makes the code more readable by making it possible to avoid multiple sets of brackets.

```
trace(masterTaskList.Wednesday[1]); // output: dentist
trace(masterTaskList.Sunday[0]);    // output: mow lawn
```

You can iterate through the task list using a `for..in` loop, but you must use bracket notation instead of dox syntax to access the value associated with each key. Because `masterTaskList` is an associative array, the elements are not necessarily retrieved in the order that you may expect, as the following example shows:

```
for (var day:String in masterTaskList)
{
  trace(day + ": " + masterTaskList[day])
}
/* output:
Sunday: mow lawn,fix chair
Wednesday: wash dishes,dentist,wash dog
Friday: wash dishes,clean house
Thursday: wash dishes
Monday: wash dishes,take out trash
Saturday: wash dishes,wash car,pay rent
Tuesday: wash dishes,pay bills
*/
```

Cloning arrays

The Array class has no built-in method for making copies of arrays. You can create a *shallow copy* of an array by calling either the `concat()` or `slice()` methods with no arguments. In a shallow copy, if the original array has elements that are objects, only the references to the objects are copied rather than the objects themselves. The copy points to the same objects as the original does. Any changes made to the objects are reflected in both arrays.

In a *deep copy*, any objects found in the original array are also copied so that the new array does not point to the same objects as does the original array. Deep copying requires more than one line of code, which usually calls for the creation of a function. Such a function could be created as a global utility function or as a method of an Array subclass.

The following example defines a function named `clone()` that does deep copying. The algorithm is borrowed from a common Java programming technique. The function creates a deep copy by serializing the array into an instance of the ByteArray class, and then reading the array back into a new array. This function accepts an object so that it can be used with both indexed arrays and associative arrays, as shown in the following code:

```
import flash.utils.ByteArray;

function clone(source:Object):*
{
  var myBA:ByteArray = new ByteArray();
  myBA.writeObject(source);
  myBA.position = 0;
  return(myBA.readObject());
}
```

Advanced Topics

Extending the Array class

The Array class is one of the few core classes that is not final, which means that you can create your own subclass of Array. This section provides an example of how to create a subclass of Array and discusses some of the issues that can arise during the process.

As mentioned previously, arrays in ActionScript are not typed, but you can create a subclass of Array that accepts elements of only a specific data type. The example in the following sections defines an Array subclass named TypedArray that limits its elements to values of the data type specified in the first parameter. The TypedArray class is presented merely as an example of how to extend the Array class and may not be suitable for production purposes for several reasons. First, type checking occurs at run time rather than at compile time. Second, when a TypedArray method encounters a mismatch, the mismatch is ignored and no exception is thrown, although the methods can be easily modified to throw exceptions. Third, the class cannot prevent the use of the array access operator to insert values of any type into the array. Fourth, the coding style favors simplicity over performance optimization.

Declaring the subclass

Use the `extends` keyword to indicate that a class is a subclass of Array. A subclass of Array should use the `dynamic` attribute, just as the Array class does. Otherwise, your subclass will not function properly.

The following code shows the definition of the TypedArray class, which contains a constant to hold the data type, a constructor method, and the four methods that are capable of adding elements to the array. The code for each method is omitted in this example, but is delineated and explained fully in the sections that follow:

```
public dynamic class TypedArray extends Array
{
  private const dataType:Class;

  public function TypedArray(...args) {}

  AS3 override function concat(...args):Array {}

  AS3 override function push(...args):uint {}

  AS3 override function splice(...args) {}

  AS3 override function unshift(...args):uint {}
}
```

The four overridden methods all use the AS3 namespace instead of the `public` attribute because this example assumes that the compiler option `-as3` is set to `true` and the compiler option `-es` is set to `false`. These are the default settings for Adobe Flex Builder 2 and for Adobe Flash CS3 Professional. For more information, see "The AS3 namespace" on page 147.

TIP

If you are an advanced developer who prefers to use prototype inheritance, you can make two minor changes to the TypedArray class to make it compile with the compiler option `-es` set to `true`. First, remove all occurrences of the `override` attribute and replace the AS3 namespace with the `public` attribute. Second, substitute `Array.prototype` for all four occurrences of `super`.

TypedArray constructor

The subclass constructor poses an interesting challenge because the constructor must accept a list of arguments of arbitrary length. The challenge is how to pass the arguments on to the superconstructor to create the array. If you pass the list of arguments as an array, the superconstructor considers it a single argument of type Array and the resulting array is always 1 element long. The traditional way to handle pass-through argument lists is to use the `Function.apply()` method, which takes an array of arguments as its second parameter but converts it to a list of arguments when executing the function. Unfortunately, the `Function.apply()` method cannot be used with constructors.

The only option left is to recreate the logic of the Array constructor in the TypedArray constructor. The following code shows the algorithm used in the Array class constructor, which you can reuse in your Array subclass constructor:

```
public dynamic class Array
{
  public function Array(...args)
  {
    var n:uint = args.length
    if (n == 1 && (args[0] is Number))
    {
      var dlen:Number = args[0];
      var ulen:uint = dlen;
      if (ulen != dlen)
      {
        throw new RangeError("Array index is not a 32-bit unsigned integer
  ("+dlen+")");
      }
      length = ulen;
    }
    else
    {
      length = n;
      for (var i:int=0; i < n; i++)
```

```
            {
              this[i] = args[i]
            }
        }
    }
}
```

The TypedArray constructor shares most of the code from the Array constructor, with only four changes to the code. First, the parameter list includes a new required parameter of type Class that allows specification of the array's data type. Second, the data type passed to the constructor is assigned to the `dataType` variable. Third, in the `else` statement, the value of the `length` property is assigned after the `for` loop so that `length` includes only arguments that are the proper type. Fourth, the body of the `for` loop uses the overridden version of the `push()` method so that only arguments of the correct data type are added to the array. The following example shows the TypedArray constructor function:

```
public dynamic class TypedArray extends Array
{
  private var dataType:Class;
  public function TypedArray(typeParam:Class, ...args)
  {
    dataType = typeParam;
    var n:uint = args.length
    if (n == 1 && (args[0] is Number))
    {
      var dlen:Number = args[0];
      var ulen:uint = dlen
      if (ulen != dlen)
      {
        throw new RangeError("Array index is not a 32-bit unsigned integer
  ("+dlen+")")
      }
      length = ulen;
    }
    else
    {
      for (var i:int=0; i < n; i++)
      {
        // type check done in push()
        this.push(args[i])
      }
      length = this.length;
    }
  }
}
```

TypedArray overridden methods

The TypedArray class overrides the four methods of the Array class that are capable of adding elements to an array. In each case, the overridden method adds a type check that prevents the addition of elements that are not the correct data type. Subsequently, each method calls the superclass version of itself.

The `push()` method iterates through the list of arguments with a `for..in` loop and does a type check on each argument. Any argument that is not the correct type is removed from the `args` array with the `splice()` method. After the `for..in` loop ends, the `args` array contains values only of type `dataType`. The superclass version of `push()` is then called with the updated `args` array, as the following code shows:

```
AS3 override function push(...args):uint
{
    for (var i:* in args)
    {
        if (!(args[i] is dataType))
        {
            args.splice(i,1);
        }
    }
    return (super.push.apply(this, args));
}
```

The `concat()` method creates a temporary TypedArray named `passArgs` to store the arguments that pass the type check. This allows the reuse of the type check code that exists in the `push()` method. A `for..in` loop iterates through the `args` array, and calls `push()` on each argument. Because `passArgs` is typed as TypedArray, the TypedArray version of `push()` is executed. The `concat()` method then calls its own superclass version, as the following code shows:

```
AS3 override function concat(...args):Array
{
    var passArgs:TypedArray = new TypedArray(dataType);
    for (var i:* in args)
    {
        // type check done in push()
        passArgs.push(args[i]);
    }
    return (super.concat.apply(this, passArgs));
}
```

The `splice()` method takes an arbitrary list of arguments, but the first two arguments always refer to an index number and the number of elements to delete. This is why the overridden `splice()` method does type checking only for `args` array elements in index positions 2 or higher. One point of interest in the code is that there appears to be a recursive call to `splice()` inside the `for` loop, but this is not a recursive call because `args` is of type Array rather than TypedArray, which means that the call to `args.splice()` is a call to the superclass version of the method. After the `for..in` loop concludes, the `args` array contains only values of the correct type in index positions 2 or higher, and `splice()` calls its own superclass version, as shown in the following code:

```
AS3 override function splice(...args):*
{
    if (args.length > 2)
    {
        for (var i:int=2; i< args.length; i++)
        {
            if (!(args[i] is dataType))
            {
                args.splice(i,1);
            }
        }
    }
    return (super.splice.apply(this, args));
}
```

The `unshift()` method, which adds elements to the beginning of an array, also accepts an arbitrary list of arguments. The overridden `unshift()` method uses an algorithm very similar to that used by the `push()` method, as shown in the following example code:

```
AS3 override function unshift(...args):uint
{
    for (var i:* in args)
    {
        if (!(args[i] is dataType))
        {
            args.splice(i,1);
        }
    }
    return (super.unshift.apply(this, args));
}
}
```

Example: PlayList

The PlayList example demonstrates techniques for working with arrays, in the context of a music playlist application that manages a list of songs. These techniques are:

- Creating an indexed array
- Adding items to an indexed array
- Sorting an array of objects by different properties, using different sorting options
- Converting an array to a character-delimited string

The PlayList application files can be found in the Samples/PlayList folder. The application consists of the following files:

File	Description
PlayListApp.mxml	The main application file in MXML for Flex.
com/example/programmingas3/playlist/PlayList.as	Provides the playlist functionality for the application, such as adding and sorting songs.
com/example/programmingas3/playlist/Song.as	A value object representing information about a single song. The items that are managed by the PlayList class are Song instances.
com/example/programmingas3/playlist/SortProperty.as	A pseudo-enumeration whose available values represent the properties of the Song class by which a list of Song objects can be sorted.

PlayList class overview

The PlayList class manages a set of Song objects. It has public methods with functionality for adding a song to the playlist (the `addSong()` method) and sorting the songs in the list (the `sortList()` method). In addition, the class includes a read-only accessor property, `songList`, which provides access to the actual set of songs in the playlist. Internally, the PlayList class keeps track of its songs using a private Array variable:

```
public class PlayList
{
  private var _songs:Array;
  private var _currentSort:SortProperty = null;
  private var _needToSort:Boolean = false;
  ...
}
```

In addition to the `_songs` Array variable, which is used by the PlayList class to keep track of its list of songs, two other private variables keep track of whether the list needs to be sorted (`_needToSort`) and which property the song list is sorted by at a given time (`_currentSort`).

As with all objects, declaring an Array instance is only half the job of creating an Array. Before accessing an Array instance's properties or methods, it must be instantiated, which is done in the PlayList class's constructor.

```
public function PlayList()
{
   this._songs = new Array();
   // Set the initial sorting.
   this.sortList(SortProperty.TITLE);
}
```

The first line of the constructor instantiates the `_songs` variable, so that it is ready to be used. In addition, the `sortList()` method is called to set the initial sort-by property.

Adding a song to the list

When a user enters a new song into the application, the code in the data entry form calls the PlayList class's `addSong()` method.

```
/**
 * Adds a song to the playlist.
 */
public function addSong(song:Song):void
{
   this._songs.push(song);
   this._needToSort = true;
}
```

Inside `addSong()`, the `_songs` array's `push()` method is called, adding the Song object that was passed to `addSong()` as a new element in that array. With the `push()` method, the new element is added to the end of the array, regardless of any sorting that might have been applied previously. This means that after the `push()` method has been called, the list of songs is likely to no longer be sorted correctly, so the `_needToSort` variable is set to `true`. In theory, the `sortList()` method could be called immediately, removing the need to keep track of whether the list is sorted or not at a given time. In practice, however, there is no need for the list of songs to be sorted until immediately before it is retrieved. By deferring the sorting operation, the application doesn't perform sorting that is unnecessary if, for example, several songs are added to the list before it is retrieved.

Sorting the list of songs

Because the Song instances that are managed by the playlist are complex objects, users of the application may wish to sort the playlist according to different properties, such as song title or year of publication. In the PlayList application, the task of sorting the list of songs has three parts: identifying the property by which the list should be sorted, indicating what sorting options need to be used when sorting by that property, and performing the actual sort operation.

Properties for sorting

A Song object keeps track of several properties, including song title, artist, publication year, filename, and a user-selected set of genres in which the song belongs. Of these, only the first three are practical for sorting. As a matter of convenience for developers, the example includes the SortProperty class, which acts as an enumeration with values representing the properties available for sorting.

```
public static const TITLE:SortProperty = new SortProperty("title");
public static const ARTIST:SortProperty = new SortProperty("artist");
public static const YEAR:SortProperty = new SortProperty("year");
```

The SortProperty class contain three constants, `TITLE`, `ARTIST`, and `YEAR`, each of which stores a String containing the actual name of the associated Song class property that can be used for sorting. Throughout the rest of the code, whenever a sort property is indicated, it is done using the enumeration member. For instance, in the PlayList constructor, the list is sorted initially by calling the `sortList()` method, as follows:

```
    // Set the initial sorting.
    this.sortList(SortProperty.TITLE);
```

Because the property for sorting is specified as `SortProperty.TITLE`, the songs are sorted according to their title.

Sorting by property and specifying sort options

The work of actually sorting the list of songs is performed by the PlayList class in the `sortList()` method, as follows:

```
/**
 * Sorts the list of songs according to the specified property.
 */
public function sortList(sortProperty:SortProperty):void
{
   ...
   var sortOptions:uint;
   switch (sortProperty)
   {
      case SortProperty.TITLE:
         sortOptions = Array.CASEINSENSITIVE;
         break;
      case SortProperty.ARTIST:
         sortOptions = Array.CASEINSENSITIVE;
         break;
      case SortProperty.YEAR:
         sortOptions = Array.NUMERIC;
         break;
   }

   // Perform the actual sorting of the data.
   this._songs.sortOn(sortProperty.propertyName, sortOptions);

   // Save the current sort property.
   this._currentSort = sortProperty;

   // Record that the list is sorted.
   this._needToSort = false;
}
```

When sorting by title or artist, it makes sense to sort alphabetically, but when sorting by year, it's most logical to perform a numeric sort. The `switch` statement is used to define the appropriate sorting option, stored in the variable `sortOptions`, according to the value specified in the `sortProperty` parameter. Here again the named enumeration members are used to distinguish between properties, rather than hard-coded values.

With the sort property and sort options determined, the `_songs` array is actually sorted by calling its `sortOn()` method, passing those two values as parameters. The current sort property is recorded, as is the fact that the song list is currently sorted.

Combining array elements into a character-delimited string

In addition to using an array to maintain the song list in the PlayList class, in this example arrays are also used in the Song class to help manage the list of genres to which a given song belongs. Consider this snippet from the Song class's definition:

```
private var _genres:String;

public function Song(title:String, artist:String, year:uint,
  filename:String, genres:Array)
{
  ...
  // Genres are passed in as an array
  // but stored as a semicolon-separated string.
  this._genres = genres.join(";");
}
```

When creating a new Song instance, the `genres` parameter that is used to specify the genre (or genres) the song belongs to is defined as an Array instance. This makes it convenient to group multiple genres together into a single variable that can be passed to the constructor. However, internally the Song class maintains the genres in the private `_genres` variable as a semicolon-separated String instance. The Array parameter is converted into a semicolon-separated string by calling its `join()` method with the literal string value `";"` as the specified delimiter.

By the same token, the `genres` accessors allow genres to be set or retrieved as an Array:

```
  public function get genres():Array
  {
    // Genres are stored as a semicolon-separated String,
    // so they need to be transformed into an Array to pass them back out.
    return this._genres.split(";");
  }
  public function set genres(value:Array):void
  {
    // Genres are passed in as an array,
    // but stored as a semicolon-separated string.
    this._genres = value.join(";");
  }
```

The `genres set` accessor behaves exactly the same as the constructor; it accepts an Array and calls the `join()` method to convert it to a semicolon-separated String. The `get` accessor performs the opposite operation: the `_genres` variable's `split()` method is called, splitting the String into an array of values using the specified delimiter (the literal string value `";"` as before).

CHAPTER 9

Handling Errors

9

To “handle” an error means you build logic into your application that responds to, or fixes, an error, generated either when an application is compiled or when a compiled application is running. When your application handles errors, *something* occurs as a response when the error is encountered, as opposed to no response and whatever process created the error failing silently. Used correctly, error handling helps shield your application and its users from otherwise unexpected behavior.

However, error handling is a broad category that includes responding to many kinds of errors that are thrown during compilation or at run time. This chapter focuses on how to handle run-time errors, the different types of errors that can be generated, and the advantages of the new error handling system in ActionScript 3.0. This chapter also explains how to implement your own custom error handling strategies for your applications.

Contents

Types of errors

When you develop and run applications, you encounter different types of errors and error terminology. The following list introduces the major error types and terms:

- *Compile-time errors* are raised by the ActionScript compiler during code compilation. Compile-time errors occur when syntactical problems in your code prevent your application from being built.
- *Run-time errors* occur when you run your application after you compile it. Run-time errors represent errors that are caused while a SWF file plays in Adobe Flash Player 9. In most cases, you will be able to handle run-time errors as they occur, reporting them to the user and taking steps to keep your application running. If the error is a fatal error, such as not being able to connect to a remote website or load required data, you can use error handling to allow your application to finish gracefully.
- *Synchronous errors* are run-time errors that occur at the time a function is invoked—for example, when you try to use a specific method and the argument you pass to the method is invalid, so Flash Player throws an exception. Most errors occur synchronously—at the time the statement executes—and the flow of control passes immediately to the most applicable `catch` statement.

  For example, the following code excerpt throws a run-time error because the `browse()` method is not called before the program attempts to upload a file:

  ```
  var fileRef:FileReference = new FileReference();
  try
  {
    fileRef.upload("http://www.yourdomain.com/fileupload.cfm");
  }
  catch (error:IllegalOperationError)
  {
    trace(error);
    // Error #2037: Functions called in incorrect sequence, or earlier
    // call was unsuccessful.
  }
  ```

  In this case, a run-time error is thrown synchronously because Flash Player determined that the `browse()` method was not called before the file upload was attempted.

  For detailed information on synchronous error handling, see "Handling synchronous errors in an application" on page 261.

- *Asynchronous errors* are run-time errors that occur at various points during run time; they generate events and are caught by event listeners. An asynchronous operation is one in which a function initiates an operation, but doesn't wait for it to complete. You can create an error event listener to wait for the application or user to try some operation, and if the operation fails, you catch the error with an event listener and respond to the error event. Then, the event listener calls an event handler function to respond to the error event in a useful manner. For example, the event handler could launch a dialog box that prompts the user to resolve the error.

  Consider the file-upload synchronous error example presented earlier. If you successfully call the `browse()` method before beginning a file upload, Flash Player would dispatch several events. For example, when an upload starts, the `open` event is dispatched. When the file upload operation completes successfully, the `complete` event is dispatched. Because event handling is asynchronous (that is, it does not occur at specific, known, predesignated times), you need to use the `addEventListener()` method to listen for these specific events, as the following code shows:

```
var fileRef:FileReference = new FileReference();
fileRef.addEventListener(Event.SELECT, selectHandler);
fileRef.addEventListener(Event.OPEN, openHandler);
fileRef.addEventListener(Event.COMPLETE, completeHandler);
fileRef.browse();

function selectHandler(event:Event):void
{
  trace("...select...");
  var request:URLRequest = new URLRequest("http://www.yourdomain.com/
  fileupload.cfm");
  request.method = URLRequestMethod.POST;
  event.target.upload(request.url);
}
function openHandler(event:Event):void
{
  trace("...open...");
}
function completeHandler(event:Event):void
{
  trace("...complete...");
}
```

  For detailed information on asynchronous error handling, see "Responding to error events and status" on page 267.

- *Uncaught exceptions* are errors thrown with no corresponding logic (like a `catch` statement) to respond to them. If your application throws an error, and no appropriate `catch` statement or event handler can be found at the current or higher level to handle the error, the error is considered an uncaught exception.

  At run time, Flash Player ignores, by design, uncaught errors and tries to continue playing if the error doesn't stop the current SWF file, because users can't necessarily resolve an error themselves. The process of ignoring an uncaught error is called "failing silently" and can complicate debugging applications. The debugger version of Flash Player responds to an uncaught error by terminating the current script and displaying the uncaught error in `trace` statement output or writing the error message to a log file. If the exception object is an instance of the Error class or one of its subclasses, the `getStackTrace()` method is invoked, and the stack trace information will also be displayed in `trace` statement output or in a log file. For more information about using debugger version of Flash Player, see "Working with the debugger version of Flash Player" on page 260.

Error handling in ActionScript 3.0

Since many applications can run without building the logic to handle errors, developers are tempted to postpone building error handling into their applications. However, without error handling, an application may easily stall or frustrate the user if something doesn't work as expected. ActionScript 2.0 has an Error class that allows you to build logic into custom functions to throw an exception with a specific message. Because error handling is critical for making a user-friendly application, ActionScript 3.0 includes an expanded architecture for catching errors.

NOTE

While the *ActionScript 3.0 Language Reference* documents the exceptions thrown by many methods, it may not include all possible exceptions for each method. A method may throw an exception for syntax errors or other problems that are not noted explicitly in the method description, even when the description does list some of the exceptions a method throws.

ActionScript 3.0 error handling elements

ActionScript 3.0 includes many tools for error handling, including:

- Error classes. In compliance with the ECMAScript (ECMA-262) edition 4 draft language specification, ActionScript 3.0 includes a broad range of Error classes to expand the scope of situations that may produce error objects. Each Error class helps applications handle and respond to specific error conditions, whether they are related to system errors (like a MemoryError condition), coding errors (like an ArgumentError condition), networking and communication errors (like a URIError condition), or other situations. For more information on each class, see "Comparing the Error classes" on page 271.
- Fewer silent failures. In earlier versions of Flash Player, errors were generated and reported only if you explicitly used the `throw` statement. For Flash Player 9, native ActionScript methods and properties throw run-time errors that allow you to handle these exceptions more effectively when they occur, and then individually react to each exception.
- Clear error messages displayed during debugging. When you are using the debugger version of Flash Player, problematic code or situations will generate robust error messages, which help you easily identify reasons why a particular block of code fails. This makes fixing errors more efficient. For more information, see "Working with the debugger version of Flash Player" on page 260.
- Precise errors allow for clear error messages displayed to users at run time. In previous versions of Flash Player, the `FileReference.upload()` method returned a Boolean value of `false` if the `upload()` call was unsuccessful, indicating one of five possible errors. If an error occurs when you call the `upload()` method in ActionScript 3.0, you can throw one of four specific errors, which helps you display more accurate error messages to end users.
- Refined error handling. Distinct errors are thrown for many common situations. For example, in ActionScript 2.0, before a FileReference object has been populated, the `name` property has the value `null` (so, before you can use or display the `name` property, you need to ensure that the value is set and not `null`). In ActionScript 3.0, if you attempt to access the `name` property before it has been populated, Flash Player throws an IllegalOperationError, which informs you that the value has not been set, and you can use `try..catch..finally` blocks to handle the error. For more information see "Using try..catch..finally statements" on page 261.
- No significant performance drawbacks. Using `try..catch..finally` blocks to handle errors takes little or no additional resources compared to previous versions of ActionScript.
- An ErrorEvent class that allows you to build listeners for specific asynchronous error events. For more information see "Responding to error events and status" on page 267.

Error handling strategies

As long as your application doesn’t encounter a problematic condition, it may still run successfully if you don’t build error handling logic into your code. However, if you don’t actively handle errors and your application does encounter a problem, your users will never know why your application fails when it does.

There are different ways you can approach error handling in your application. The following list summarizes the three major options for handling errors:

- Use `try..catch..finally` statements. These will catch synchronous errors as they occur. You can nest your statements into a hierarchy to catch exceptions at various levels of code execution. For more information, see “Using try..catch..finally statements” on page 261.
- Create your own custom error objects. You can use the Error class to create your own custom error objects to track specific operations in your application that are not covered by built-in error types. Then you can use `try..catch..finally` statements on your custom error objects. For more information see “Creating custom error classes” on page 266.
- Write event listeners and handlers to respond to error events. By using this strategy, you can create global error handlers that let you handle similar events without duplicating a lot of code in `try..catch..finally` blocks. You are also more likely to catch asynchronous errors using this approach. For more information, see “Responding to error events and status” on page 267.

Working with the debugger version of Flash Player

Adobe provides developers with a special edition of the Flash Player to assist debugging efforts. You obtain a copy of the debugger version of Flash Player when you install Adobe Flash CS3 or Adobe Flex Builder 2.

There is a notable difference in how debugger version and the release version of Flash Player indicate errors. The debugger version shows the error type (such as a generic Error, IOError, or EOFError), error number, and a human-readable error message. The release version shows only the error type and error number. For example, consider the following code:

```
try
{
  tf.text = myByteArray.readBoolean();
}
catch (error:EOFError)
```

```
{
  tf.text = error.toString();
}
```

If the `readBoolean()` method threw an EOFError in the debugger version of Flash Player, the following message would be displayed in the `tf` text field: "EOFError: Error #2030: End of file was encountered."

The same code in a release version of Flash Player would display the following text: "EOFError: Error #2030."

In order to keep Flash Player's resources and size to a minimum in the release version, error message strings are not present. You can look up the error number in the documentation (the appendixes of the *ActionScript 3.0 Language Reference*) to correlate to an error message. Alternatively, you can reproduce the error using the debugger version of Flash Player to see the full message.

Handling synchronous errors in an application

The most common error handling is synchronous error handling logic, where you insert statments into your code to catch synchronous errors at run time. This type of error handling lets your application notice and recover from run-time errors when functions fail. The logic for catching a synchronous error includes `try..catch..finally` statements, which literally try an operation, catch any error response from Flash Player, and finally execute some other operation to handle the failed operation.

Using try..catch..finally statements

When you work with synchronous run-time errors, use the `try..catch..finally` statements to catch errors. When a run-time error occurs, Flash Player throws an exception, which means that Flash Player suspends normal execution and creates a special object of type Error. The Error object is then thrown to the first available `catch` block.

The `try` statement encloses statements that have the potential to create errors. You always use the `catch` statement with a `try` statement. If an error is detected in one of the statements in the `try` statement block, the `catch` statements that are attached to that `try` statement will execute.

The `finally` statement encloses statements that will execute whether or not an error occurs in the `try` block. If there is no error, the statements within the `finally` block execute after the `try` block statements complete. If there is an error, the appropriate `catch` statement executes first, followed by the statements in the `finally` block.

The following code demonstrates the syntax for using the `try..catch..finally` statements:

```
try
{
   // some code that could throw an error
}
catch (err:Error)
{
   // code to react to the error
}
finally
{
   // Code that runs whether or not an error was thrown. This code can clean
  // up after the error, or take steps to keep the application running.
}
```

Each `catch` statement identifies a specific type of exception that it handles. The `catch` statement can specify only error classes that are subclasses of the Error class. Each `catch` statement is checked in order. Only the first `catch` statement that matches the type of error thrown will execute. In other words, if you first check the higher-level Error class and then a subclass of the Error class, only the higher-level Error class will match. The following code illustrates this point:

```
try
{
  throw new ArgumentError("I am an ArgumentError");
}
catch (error:Error)
{
  trace("<Error> " + error.message);
}
catch (error:ArgumentError)
{
  trace("<ArgumentError> " + error.message);
}
```

The previous code displays the following output:

```
<Error> I am an ArgumentError
```

In order to correctly catch the ArgumentError, you need to make sure that the most specific error types are listed first, and the more generic error types are listed later, as the following code shows:

```
try
{
    throw new ArgumentError("I am an ArgumentError");
}
catch (error:ArgumentError)
{
    trace("<ArgumentError> " + error.message);
}
catch (error:Error)
{
    trace("<Error> " + error.message);
}
```

Several methods and properties in the Flash Player API throw run-time errors if they encounter errors while they execute. For example, the `close()` method in the Sound class throws an IOError if the method is unable to close the audio stream, as demonstrated in the following code:

```
var mySound:Sound = new Sound();
try
{
    mySound.close();
}
catch (error:IOError)
{
    // Error #2029: This URLStream object does not have an open stream.
}
```

As you become more familiar with the *ActionScript 3.0 Language Reference*, you'll notice which methods throw exceptions, as detailed in each method's description.

The throw statement

Flash Player throws exceptions when it encounters errors in your application at run time. In addition, you can explicitly throw exceptions yourself using the `throw` statement. When explicitly throwing errors, Adobe recommends that you throw instances of the Error class or its subclasses. The following code demonstrates a `throw` statement that throws an instance of the Error class, `MyErr`, and eventually calls a function, `myFunction()`, to respond after the error is thrown:

```
var MyError:Error = new Error("Encountered an error with the numUsers
    value", 99);
var numUsers:uint = 0;
try
{
    if (numUsers == 0)
    {
        trace("numUsers equals 0");
    }
}
```

```
catch (error:uint)
{
  throw MyError; // Catch unsigned integer errors.
}
catch (error:int)
{
  throw MyError; // Catch integer errors.
}
catch (error:Number)
{
  throw MyError; // Catch number errors.
}
catch (error:*)
{
  throw MyError; // Catch any other error.
}
finally
{
  myFunction(); // Perform any necessary cleanup here.
}
```

Notice that the `catch` statements are ordered so that the most specific data types are listed first. If the `catch` statement for the Number data type was listed first, neither the `catch` statement for the uint data type nor the catch statement for the int data type would ever get executed.

NOTE

In the Java programming language, each function that can throw an exception must declare this fact, listing the exception classes it can throw in a `throws` clause attached to the function declaration. ActionScript does not require you to declare the exceptions that can be thrown by a function.

Displaying a simple error message

One of the biggest benefits of the new exception and error event model is that it allows you to tell users when and why an action has failed. Your part is to write the code to display the message and offer options in response.

The following code shows a simple `try..catch` statement to display the error in a text field:

```
package
{
  import flash.display.Sprite;
  import flash.text.TextField;

  public class SimpleError extends Sprite
  {
    public var employee:XML =
      <EmpCode>
        <costCenter>1234</costCenter>
```

```
            <costCenter>1-234</costCenter>
        </EmpCode>;

    public function SimpleError()
    {
        try
        {
            if (employee.costCenter.length() != 1)
            {
                throw new Error("Error, employee must have exactly one cost 
  center assigned.");
            }
        }
        catch (error:Error)
        {
            var errorMessage:TextField = new TextField();
            errorMessage.autoSize = TextFieldAutoSize.LEFT;
            errorMessage.textColor = 0xFF0000;
            errorMessage.text = error.message;
            addChild(errorMessage);
        }
    }
  }
}
```

Using a wider range of error classes and built-in compiler errors, ActionScript 3.0 offers more information than previous versions of ActionScript about why something has failed. This enables you to build more stable applications with better error handling.

Rethrowing errors

When you build applications, there are several occasions in which you may need to rethrow an error if you are unable to handle the error properly. For example, the following code shows a nested `try..catch` block, which rethrows a custom ApplicationError if the nested `catch` block is unable to handle the error:

```
try
{
  try
  {
    trace("<< try >>");
    throw new ArgumentError("some error which will be rethrown");
  }
  catch (error:ApplicationError)
  {
    trace("<< catch >> " + error);
    trace("<< throw >>");
    throw error;
  }
  catch (error:Error)
  {
    trace("<< Error >> " + error);
  }
}
catch (error:ApplicationError)
{
  trace("<< catch >> " + error);
}
```

The output from the previous snippet would be the following:

```
<< try >>
<< catch >> ApplicationError: some error which will be rethrown
<< throw >>
<< catch >> ApplicationError: some error which will be rethrown
```

The nested `try` block throws a custom ApplicationError error that is caught by the subsequent `catch` block. This nested `catch` block can try to handle the error, and if unsuccessful, throw the ApplicationError object to the enclosing `try..catch` block.

Creating custom error classes

You can extend one of the standard Error classes to create your own specialized error classes in ActionScript. There are a number of reasons to create your own error classes:

- To identify specific errors or groups of errors that are unique to your application.

  For example, you may want to take different actions for errors thrown by your own code, in addition to those trapped by Flash Player. You can create a subclass of the Error class to track the new error data type in `try..catch` blocks.
- To provide unique error display capabilities for errors generated by your application.

  For example, you can create a new `toString()` method that formats your error messages in a certain way. You can also define a `lookupErrorString()` method that takes an error code and retrieves the proper message based on the user's language preference.

A specialized error class must extend the core ActionScript Error class. Here is an example of a specialized AppError class that extends the Error class:

```
public class AppError extends Error
{
  public function AppError(message:String, errorID:int)
  {
    super(message, errorID);
  }
}
```

The following shows an example of using AppError in your project:

```
try
{
  throw new AppError("Encountered Custom AppError", 29);
}
catch (error:AppError)
{
  trace(error.errorID + ": " + error.message)
}
```

NOTE

If you want to override the `Error.toString()` method in your subclass, you need to give it one ...(rest) parameter. The ECMAScript (ECMA-262) edition 3 language specification defines the `Error.toString()` method that way, and ActionScript 3.0 defines it the same way for backward compatibility with that specification. Therefore, when you override the `Error.toString()` method, you must match the parameters exactly. You will not want to pass any parameters to your `toString()` method at run time, because those parameters are ignored.

Responding to error events and status

One of the most noticeable improvements to error handling in ActionScript 3.0 is the support for error event handling for responding to asynchronous run-time errors. (For a definition of asynchronous errors, see "Types of errors" on page 256.)

You can create event listeners and event handlers to respond to the error events. Many classes dispatch error events the same way they dispatch other events. For example, an instance of the XMLSocket class normally dispatches three types of events: `Event.CLOSE`, `Event.CONNECT`, and `DataEvent.DATA`. However, when a problem occurs, the XMLSocket class can dispatch the `IOErrorEvent.IOError` or the `SecurityErrorEvent.SECURITY_ERROR`. For more information about event listeners and event handlers, see Chapter 13, "Handling Events," on page 345.

Error events fit into one of two categories:

- Error events that extend the ErrorEvent class

  The flash.events.ErrorEvent class contains the properties and methods for managing Flash Player run-time errors related to networking and communication operations. The AsyncErrorEvent, IOErrorEvent, and SecurityErrorEvent classes extend the ErrorEvent class. If you're using the debugger version of Flash Player, a dialog box will inform you at run-time of any error events without listener functions that the player encounters.

- Status-based error events

  The status-based error events are related to the `netStatus` and `status` properties of the networking and communication classes. If Flash Player encounters a problem when reading or writing data, the value of the `netStatus.info.level` or `status.level` properties (depending on the class object you're using) is set to the value `"error"`. You respond to this error by checking if the `level` property contains the value `"error"` in your event handler function.

Working with error events

The ErrorEvent class and its subclasses contain error types for handling errors dispatched by Flash Player as it tries to read or write data.

The following example uses both a `try..catch` statement and error event handlers to display any errors detected while trying to read a local file. You can add more sophisticated handling code to provide a user with options or otherwise handle the error automatically in the places indicated by the comment "your error handling code here":

```
package
{
  import flash.display.Sprite;
  import flash.errors.IOError;
  import flash.events.IOErrorEvent;
  import flash.events.TextEvent;
  import flash.media.Sound;
  import flash.media.SoundChannel;
  import flash.net.URLRequest;
  import flash.text.TextField;

  public class LinkEventExample extends Sprite
  {
    private var myMP3:Sound;
    public function LinkEventExample()
    {
      myMP3 = new Sound();
      var list:TextField = new TextField();
      list.autoSize = TextFieldAutoSize.LEFT;
      list.multiline = true;
      list.htmlText = "<a href=\"event:track1.mp3\">Track 1</a><br>";
      list.htmlText += "<a href=\"event:track2.mp3\">Track 2</a><br>";
      addEventListener(TextEvent.LINK, linkHandler);
      addChild(list);
    }

    private function playMP3(mp3:String):void
    {
      try
```

```
        {
          myMP3.load(new URLRequest(mp3));
          myMP3.play();
        }
        catch(err:Error)
        {
          trace(err.message);
          // your error handling code here
        }
        myMP3.addEventListener(IOErrorEvent.IO_ERROR, errorHandler);
      }

      private function linkHandler(linkEvent:TextEvent):void
      {
        playMP3(linkEvent.text);
        // your error handling code here
      }

      private function errorHandler(errorEvent:IOErrorEvent):void
      {
        trace(errorEvent.text);
        // your error handling code here
      }
  }
}
```

Working with status change events

Flash Player dynamically changes the value of the `netStatus.info.level` or `status.level` properties for the classes that support the `level` property. The classes that have the `netStatus.info.level` property are NetConnection, NetStream, and SharedObject. The classes that have the `status.level` property are HTTPStatusEvent, Camera, Microphone, and LocalConnection. You can write a handler function to respond to the change in `level` value and track communication errors.

The following example uses a `netStatusHandler()` function to test the value of the `level` property. If the `level` property indicates that an error has been encountered, the code traces the message "Video stream failed".

```
package
{
  import flash.display.Sprite;
  import flash.events.NetStatusEvent;
  import flash.events.SecurityErrorEvent;
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  public class VideoExample extends Sprite
  {
    private var videoUrl:String = "Video.flv";
    private var connection:NetConnection;
    private var stream:NetStream;

    public function VideoExample()
    {
      connection = new NetConnection();
      connection.addEventListener(NetStatusEvent.NET_STATUS,
  netStatusHandler);
      connection.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
  securityErrorHandler);
      connection.connect(null);
    }

    private function netStatusHandler(event:NetStatusEvent):void
    {
      if (event.info.level = "error")
      {
        trace("Video stream failed")
      }
      else
      {
        connectStream();
      }
    }

    private function securityErrorHandler(event:SecurityErrorEvent):void
    {
      trace("securityErrorHandler: " + event);
    }

    private function connectStream():void
    {
      var stream:NetStream = new NetStream(connection);
      var video:Video = new Video();
      video.attachNetStream(stream);
      stream.play(videoUrl);
      addChild(video);
    }
  }
}
```

Comparing the Error classes

ActionScript provides a number of predefined Error classes. Many of these classes are used by Flash Player, but you can also use the same Error classes in your own code. There are two main types of Error classes in ActionScript 3.0: ActionScript core Error classes and flash.error package Error classes. The core Error classes are prescribed by the ECMAScript (ECMA-262) edition 4 draft language specification. The flash.error package contents are additional classes introduced to aid ActionScript 3.0 application development and debugging.

ECMAScript core Error classes

The ECMAScript core error classes include the Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError, and URIError classes. Each of these classes are located in the top-level namespace.

Class name	Description	Notes
Error	The Error class can be used for throwing exceptions, and is the base class for the other exception classes defined in ECMAScript: EvalError, RangeError, ReferenceError, SyntaxError, TypeError, and URIError.	The Error class serves as the base class for all run-time errors thrown by Flash Player, and is the recommended base class for any custom error classes.
EvalError	An EvalError exception is thrown if any parameters are passed to the Function class's constructor or if user code calls the `eval()` function.	In ActionScript 3.0, support for the `eval()` function has been removed and attempts to use the function cause an error to be thrown. Earlier versions of Flash Player used the `eval()` function to access variables, properties, objects, or movie clips by name.
RangeError	A RangeError exception is thrown if a numeric value falls outside of an acceptable range.	For example, a RangeError would be thrown by the Timer class if a delay was either negative or was not finite. A RangeError could also be thrown if you attempted to add a display object at an invalid depth.

Class name	Description	Notes
ReferenceError	A ReferenceError exception is thrown when a reference to an undefined property is attempted on a sealed (nondynamic) object. Versions of the ActionScript compiler before ActionScript 3.0 did not throw an error when access was attempted to a property that was `undefined`. However, because the new ECMAScript specification specifies that an error should be thrown in this condition, ActionScript 3.0 throws the ReferenceError exception.	Exceptions for undefined variables point to potential bugs, helping you improve software quality. However, if you are not used to having to initialize your variables, this new ActionScript behavior may require some changes in your coding habits.
SyntaxError	A SyntaxError exception is thrown when a parsing error occurs in your ActionScript code. For more information, see Section 15.11.6.4 of the ECMAScript (ECMA-262) edition 3 (until edition 4 is available) language specification at www.ecma-international.org/publications/standards/Ecma-262.htm, as well as Section 10.3.1 of the ECMAScript for XML (E4X) specification (ECMA-357 edition 2) at www.ecma-international.org/publications/standards/Ecma-357.htm.	A SyntaxError can be thrown under the following circumstances: • ActionScript throws SyntaxError exceptions when an invalid regular expression is parsed by the RegExp class. • ActionScript throws SyntaxError exceptions when invalid XML is parsed by the XMLDocument class.

Class name	Description	Notes
TypeError	The TypeError exception is thrown when the actual type of an operand is different from the expected type. For more information, see Section 15.11.6.5 of the ECMAScript specification at www.ecma-international.org/publications/standards/Ecma-262.htm, as well as Section 10.3 of the E4X specification at www.ecma-international.org/publications/standards/Ecma-357.htm.	A TypeError can be thrown under the following circumstances: • An actual parameter of a function or method could not be coerced to the formal parameter type. • A value is assigned to a variable and cannot be coerced to the variable's type. • The right side of the `is` or `instanceof` operator is not a valid type. • The `super` keyword is used illegally. • A property lookup results in more than one binding, and is therefore ambiguous. • A method is invoked on an incompatible object. For example, a TypeError exception is thrown if a method in the RegExp class is "grafted" onto a generic object and then invoked.
URIError	The URIError exception is thrown when one of the global URI handling functions is used in a way that is incompatible with its definition. For more information, see Section 15.11.6.6 of the ECMAScript specification at www.ecma-international.org/publications/standards/Ecma-262.htm.	A URIError can be thrown under the following circumstances: An invalid URI is specified for a Flash Player API function that expects a valid URI, such as `Socket.connect()`.

ActionScript core Error classes

In addition to the core ECMAScript Error classes, ActionScript adds several classes of its own for ActionScript-specific error conditions and error handling functionality.

Because these classes are ActionScript language extensions to ECMAScript edition 4 draft language specification that could potentially be interesting additions to the draft specification, they are kept at the top level instead of being placed in a package like flash.error.

Class name	Description	Notes
ArgumentError	The ArgumentError class represents an error that occurs when the parameter values supplied during a function call do not match the parameters defined for that function.	Some examples of argument errors include the following: • Too few or too many arguments are supplied to a method. • An argument was expected to be a member of an enumeration, and was not.
SecurityError	The SecurityError exception is thrown when a security violation takes place and access is denied.	Some examples of security errors include the following: • An unauthorized property access or method call is made across a security sandbox boundary. • An attempt was made to access a URL not permitted by the security sandbox. • A socket connection was attempted to an unauthorized port number—for example, a port below 1024—without a policy file present. • An attempt was made to access the user's camera or microphone, and the request to access the device was denied by the user.
VerifyError	A VerifyError exception is thrown when a malformed or corrupted SWF file is encountered.	When a SWF file loads another SWF file, the parent SWF can catch a VerifyError generated by the loaded SWF.

flash.error package Error classes

The flash.error package contains Error classes that are considered part of the Flash Player API. In contrast to the Error classes just described, the flash.error package communicates errors events that are specific to Flash Player.

Class name	Description	Notes
EOFError	An EOFError exception is thrown when you attempt to read past the end of the available data.	For example, an EOFError is thrown when one of the read methods in the IDataInput interface is called and there is insufficient data to satisfy the read request.
IllegalOperationError	An IllegalOperationError exception is thrown when a method is not implemented or the implementation doesn't cover the current usage.	Examples of illegal operation error exceptions include the following: • A base class, such as DisplayObjectContainer, provides more functionality than the Stage can support. For example, if you attempt to get or set a mask on the Stage (using `stage.mask`), Flash Player will throw an IllegalOperationError with the message "The Stage class does not implement this property or method." • A subclass inherits a method it does not require and does not want to support. • Certain accessibility methods are called when Flash Player is compiled without accessibility support. • Authoring-only features are invoked from a run-time version of Flash Player. • You attempt to set the name of an object placed on the timeline.
IOError	An IOError exception is thrown when some type of I/O exception occurs.	You get this error, for example, when a read-write operation is attempted on a socket that is not connected or that has become disconnected.

Class name	Description	Notes
MemoryError	A MemoryError exception is thrown when a memory allocation request fails.	By default, ActionScript Virtual Machine 2 does not impose a limit on how much memory an ActionScript program may allocate. On a desktop PC, memory allocation failures are infrequent. You see an error thrown when the system is unable to allocate the memory required for an operation. So, on a desktop PC, this exception is rare unless an allocation request is extremely large; for example, a request for 3 billion bytes is impossible because a 32-bit Windows program can access only 2 GB of address space.
ScriptTimeoutError	A ScriptTimeoutError exception is thrown when a script timeout interval of 15 seconds is reached. By catching a ScriptTimeoutError exception, you can handle the script timeout more gracefully. If there is no exception handler, the uncaught exception handler will display a dialog box with an error message.	To prevent a malicious developer from catching the exception and staying in an infinite loop, only the first ScriptTimeoutError exception thrown in the course of a particular script can be caught. A subsequent ScriptTimeoutError exception cannot be caught by your code and will immediately go to the uncaught exception handler.
StackOverflowError	The StackOverflowError exception is thrown when the stack available to the script has been exhausted.	A StackOverflowError exception might indicate that infinite recursion has occurred.

Example: CustomErrors application

The CustomErrors application demonstrates techniques for working with custom errors when building an application. These techniques are:

- Validating an XML packet
- Writing a custom error
- Throwing custom errors
- Notifying users when an error is thrown

The CustomErrors application files can be found in the Samples/CustomError folder. The application consists of the following files:

File	Description
CustomErrors.mxml	The main application file consisting of the MXML user interface.
com/example/programmingas3/errors/ ApplicationError.as	A class that serves as the base error class for both the FatalError and WarningError classes.
com/example/programmingas3/errors/ FatalError.as	A class that defines a FatalError error that can be thrown by the application. This class extends the custom ApplicationError class.
com/example/programmingas3/errors/ Validator.as	A class that defines a single method that validates a user-supplied employee XML packet.
com/example/programmingas3/errors/ WarningError.as	A class that defines a WarningError error that can be thrown by the application. This class extends the custom ApplicationError class.

CustomErrors application overview

The CustomErrors.mxml file contains the user interface and some logic for the custom error application. Once the application's `creationComplete` event is dispatched, the `initApp()` method is invoked. This method defines a sample XML packet that will be verified by the Validator class. The following code shows the `initApp()` method:

```
private function initApp():void
{
  employeeXML = 
    <employee id="12345">
      <firstName>John</firstName>
      <lastName>Doe</lastName>
      <costCenter>12345</costCenter>
      <costCenter>67890</costCenter>
    </employee>;
}
```

The XML packet is later displayed in a TextArea component instance on the Stage. This allows you to modify the XML packet before attempting to revalidate it.

When the user clicks the Validate button, the `validateData()` method is called. This method validates the employee XML packet using the `validateEmployeeXML()` method in the Validator class. The following code shows the `validateData()` method:

```
public function validateData():void
{
  try
  {
    var tempXML:XML = XML(xmlText.text);
    Validator.validateEmployeeXML(tempXML);
    status.text = "The XML was successfully validated.";
  }
  catch (error:FatalError)
  {
    showFatalError(error);
  }
  catch (error:WarningError)
  {
    showWarningError(error);
  }
  catch (error:Error)
  {
    showGenericError(error);
  }
}
```

First, a temporary XML object is created using the contents of the TextArea component instance `xmlText`. Next, the `validateEmployeeXML()` method in the custom Validator class (com.example.programmingas3/errors/Validator.as) is invoked and passes the temporary XML object as a parameter. If the XML packet is valid, the `status` Label component instance displays a success message and the application exits. If the `validateEmployeeXML()` method threw a custom error (that is, a FatalError, WarningError, or a generic Error occurred), the appropriate `catch` statement executes and calls either the `showFatalError()`, `showWarningError()`, or `showGenericError()` methods. Each of these methods displays an appropriate message in an Alert component to notify the user of the specific error that occurred. Each method also updates the `status` Label component instance with a specific message.

If a fatal error occurs during an attempt to validate the employee XML packet, the error message is displayed in an Alert component, and the `xmlText` TextArea component instance and `validateBtn` Button component instance are disabled, as the following code shows:

```
public function showFatalError(error:FatalError):void
{
  var message:String = error.message + "\n\n" + "Click OK to end.";
  var title:String = error.getTitle();
  Alert.show(message, title);
  status.text = "This application has ended.";
  this.xmlText.enabled = false;
  this.validateBtn.enabled = false;
}
```

If a warning error instead of a fatal error occurs, the error message is displayed in an Alert component instance, but the TextField and Button component instances aren't disabled. The `showWarningError()` method displays the custom error message in the Alert component instance. The message also asks the user to decide if they want to proceed with validating the XML or abort the script. The following excerpt shows the `showWarningError()` method:

```
public function showWarningError(error:WarningError):void
{
  var message:String = error.message + "\n\n" + "Do you want to exit this
  application?";
  var title:String = error.getTitle();
  Alert.show(message, title, Alert.YES | Alert.NO, null, closeHandler);
  status.text = message;
}
```

When the user closes the Alert component instance by using either the Yes or No button, the `closeHandler()` method is invoked. The following excerpt shows the `closeHandler()` method:

```
private function closeHandler(event:CloseEvent):void
{
  switch (event.detail)
  {
    case Alert.YES:
      showFatalError(new FatalError(9999));
      break;
    case Alert.NO:
      break;
  }
}
```

If the user chooses to abort the script by clicking Yes in the warning error Alert dialog, a FatalError is thrown, causing the application to terminate.

Building a custom validator

The custom Validator class contains a single method, `validateEmployeeXML()`. The `validateEmployeeXML()` method takes a single argument, `employee`, which is the XML packet that you wish to validate. The `validateEmployeeXML()` method is as follows:

```
public static function validateEmployeeXML(employee:XML):void
{
  // checks for the integrity of items in the XML
  if (employee.costCenter.length() < 1)
  {
    throw new FatalError(9000);
  }
  if (employee.costCenter.length() > 1)
  {
    throw new WarningError(9001);
  }
  if (employee.ssn.length() != 1)
  {
    throw new FatalError(9002);
  }
}
```

To be validated, an employee must belong to one (and only one) cost center. If the employee doesn't belong to any cost centers, the method throws a FatalError, which bubbles up to the `validateData()` method in the main application file. If the employee belongs to more than one cost center, a WarningError is thrown. The final check in the XML validator is that the user has exactly one social security number defined (the `ssn` node in the XML packet). If there is not exactly one `ssn` node, a FatalError error is thrown.

You can add additional checks to the `validateEmployeeXML()` method—for instance, to ensure that the `ssn` node contains a valid number, or that the employee has at least one phone number and e-mail address defined, and that both values are valid. You can also modify the XML so that each employee has a unique ID and specifies the ID of their manager.

Defining the ApplicationError class

The ApplicationError class serves as the base class for both the FatalError and WarningError classes. The ApplicationError class extends the Error class, and defines its own custom methods and properties, including defining an error ID, severity, and an XML object that contains the custom error codes and messages. This class also defines two static constants that are used to define the severity of each error type.

The ApplicationError class's constructor method is as follows:

```
public function ApplicationError()
{
  messages =
    <errors>
      <error code="9000">
        <![CDATA[Employee must be assigned to a cost center.]]>
      </error>
      <error code="9001">
        <![CDATA[Employee must be assigned to only one cost center.]]>
      </error>
      <error code="9002">
        <![CDATA[Employee must have one and only one SSN.]]>
      </error>
      <error code="9999">
        <![CDATA[The application has been stopped.]]>
      </error>
    </errors>;
}
```

Each error node in the XML object contains a unique numeric code and an error message. Error messages can be easily looked up by their error code using E4X, as seen in the following `getMessageText()` method:

```
public function getMessageText(id:int):String
{
  var message:XMLList = messages.error.(@code == id);
  return message[0].text();
}
```

The `getMessageText()` method takes a single integer argument, `id`, and returns a string. The `id` argument is the error code for the error to look up. For example, passing an `id` of 9001 retrieves the error saying that employees must be assigned to only one cost center. If more than one error has the same error code, ActionScript returns the error message only for the first result found (`message[0]` in the returned XMLList object).

The next method in this class, `getTitle()`, doesn't take any parameters and returns a string value that contains the error ID for this specific error. This value is used in the Alert component's title to help you easily identify the exact error that occurred during validation of the XML packet. The following excerpt shows the `getTitle()` method:

```
public function getTitle():String
{
  return "Error #" + id;
}
```

The final method in the ApplicationError class is `toString()`. This method overrides the function defined in the Error class so that you can customize the presentation of the error message. The method returns a string that identifies the specific error number and message that occurred.

```
public override function toString():String
{
  return "[APPLICATION ERROR #" + id + "] " + message;
}
```

Defining the FatalError class

The FatalError class extends the custom ApplicationError class and defines three methods: the FatalError constructor, `getTitle()`, and `toString()`. The first method, the FatalError constructor, takes a single integer argument, `errorID`, and sets the error's severity using the static constant values defined in the ApplicationError class, and gets the specific error's error message by calling the `getMessageText()` method in the ApplicationError class. The FatalError constructor is as follows:

```
public function FatalError(errorID:int)
{
  id = errorID;
  severity = ApplicationError.FATAL;
  message = getMessageText(errorID);
}
```

The next method in the FatalError class, `getTitle()`, overrides the `getTitle()` method defined earlier in the ApplicationError class, and appends the text "-- FATAL" in the title to inform the user that a fatal error has occurred. The `getTitle()` method is as follows:

```
public override function getTitle():String
{
  return "Error #" + id + " -- FATAL";
}
```

The final method in this class, `toString()`, overrides the `toString()` method defined in the ApplicationError class. The `toString()` method is

```
public override function toString():String
{
  return "[FATAL ERROR #" + id + "] " + message;
}
```

Defining the WarningError class

The WarningError class extends the ApplicationError class and is nearly identical to the FatalError class, except for a couple minor string changes and sets the error severity to ApplicationError.WARNING instead of ApplicationError.FATAL, as seen in the following code:

```
public function WarningError(errorID:int)
{
  id = errorID;
  severity = ApplicationError.WARNING;
  message = super.getMessageText(errorID);
}
```

CHAPTER 10

Using Regular Expressions

10

A regular expression describes a pattern that is used to find and manipulate matching text in strings. Regular expressions resemble strings, but they can include special codes to describe patterns and repetition. For example, the following regular expression matches a string that starts with the character A followed by one or more sequential digits:

```
/A\d+/
```

Regular expression patterns can be complex, and sometimes cryptic in appearance, such as the following expression to match a valid e-mail address:

```
/([0-9a-zA-Z]+[-._+&])*[0-9a-zA-Z]+@([-0-9a-zA-Z]+[.])+[a-zA-Z]{2,6}/
```

This chapter describes the basic syntax for constructing regular expressions. However, regular expressions can have many complexities and nuances. You can find detailed resources on regular expressions on the web and in bookstores. Keep in mind that different programming environments implement regular expressions in different ways. ActionScript 3.0 implements regular expressions as defined in the ECMAScript edition 3 language specification (ECMA-262).

You can use regular expressions with the following methods of the String class: `match()`, `replace()`, and `search()`. For more information on these methods, see "Finding patterns in strings and replacing substrings" on page 216.

Contents

Introduction to Regular Expressions

A regular expression describes a pattern of characters. Regular expressions are typically used to verify that a text value conforms to a particular pattern (such as verifying that a user-entered phone number has the proper number of digits) or to replace portions of a text value which match a particular pattern. For example, the following regular expression defines the pattern consisting of the letters A, B, and C in sequence:

```
/ABC/
```

Note that the regular expression literal is delineated with the forward slash (`/`) character.

Generally, you use regular expressions that match more complicated patterns than a simple string of characters. For example, the following regular expression defines the pattern consisting of the letters A, B, and C in sequence followed by any digit:

```
/ABC\d/
```

The `\d` code represents "any digit." The backslash (`\`) character is called the escape character, and combined with the character that follows it (in this case the letter d), it has special meaning in the regular expression. This chapter describes these escape character sequences and other regular expression syntax features.

The following regular expression defines the pattern of the letters ABC followed by any number of digits (note the asterisk):

```
/ABC\d*/
```

The asterisk character (`*`) is a *metacharacter*. A metacharacter is a character that has special meaning in regular expressions. The asterisk is a specific type of metacharacter called a *quantifier,* which is used to quantify the amount of repetition of a character or group of characters. For more information, see "Quantifiers" on page 293.

In addition to its pattern, a regular expression can contain flags, which specify how the regular expression is to be matched. For example, the following regular expression uses the `i` flag, which specifies that the regular expression ignores case sensitivity in matching strings:

```
/ABC\d*/i
```

For more information, see "Flags and properties" on page 299.

To search for patterns in strings and to replace characters, you can use regular expressions as parameters for methods of the String class. For example:

```
var pattern:RegExp = /\d+/; // matches one or more digits in a sequence
var str:String = "Test: 337, 4, or 57.33.";
trace(str.search(pattern)); // 6

trace(str.match(pattern)); // 337

var pattern:RegExp = /\d+/g; // The g flag makes the match global.
```

```
trace(str.match(pattern)); // 337,4, 57, 33
```

The following methods of the String class take regular expressions as parameters: `match()`, `replace()`, `search()`, and `split()`. For more information on these methods, see "Finding patterns in strings and replacing substrings" on page 216.

The RegExp class includes the following methods: `test()` and `exec()`. For more information, see "Methods for using regular expressions with strings" on page 303.

Regular expression syntax

This section describes all of the elements of ActionScript regular expression syntax.

Creating an instance of a regular expression

There are two ways to create a regular expression instance. One way uses forward slash characters (`/`) to delineate the regular expression; the other uses the `new` constructor. For example, the following regular expressions are equivalent:

```
var pattern1:RegExp = /bob/i;
var pattern2:RegExp = new RegExp("bob", "i");
```

Forward slashes delineate a regular expression literal in the same way as quotation marks delineate a string literal. The part of the regular expression within the forward slashes defines the *pattern.* The regular expression can also include *flags* after the final delineating slash. These flags are considered to be part of the regular expression, but they are separate from its pattern.

When using the `new` constructor, you use two strings to define the regular expression. The first string defines the pattern, and the second string defines the flags, as in the following example:

```
var pattern2:RegExp = new RegExp("bob", "i");
```

When including a forward slash *within* a regular expression that is defined by using the forward slash delineators, you must precede the forward slash with the backslash (`\`) escape character. For example, the following regular expression matches the pattern `1/2`:

```
var pattern:RegExp = /1\/2/;
```

To include quotation marks *within* a regular expression that is defined with the `new` constructor, you must add backslash (`\`) escape character before the quotation marks (just as you would when defining any String literal). For example, the following regular expressions match the pattern `eat at "joe's"`:

```
var pattern1:RegExp = new RegExp("eat at \"joe's\"", "");
var pattern2:RegExp = new RegExp('eat at "joe\'s"', "");
```

Do not use the backslash escape character with quotation marks in regular expressions that are defined by using the forward slash delineators. Similarly, do not use the escape character with forward slashes in regular expressions that are defined with the `new` constructor. The following regular expressions are equivalent, and they define the pattern `1/2 "joe's"`:

```
var pattern1:RegExp = /1\/2 "joe's"/;
var pattern2:RegExp = new RegExp("1/2 \"joe's\"", "");
var pattern3:RegExp = new RegExp('1/2 "joe\'s"', '');
```

Also, in a regular expression that is defined with the `new` constructor, to use a metasequence that begins with the backslash (\) character, such as `\d` (which matches any digit), type the backslash character twice:

```
var pattern:RegExp = new RegExp("\\d+", ""); // matches one or more digits
```

You must type the backlash character twice in this case, because the first parameter of the `RegExp()` constructor method is a string, and in a string literal you must type a backslash character twice to have it recognized as a single backslash character.

The sections that follow describe syntax for defining regular expression patterns.

For more information on flags, see "Flags and properties" on page 299.

Characters, metacharacters, and metasequences

The simplest regular expression is one that matches a sequence of characters, as in the following example:

```
var pattern:RegExp = /hello/;
```

However, the following characters, known as metacharacters, have special meanings in regular expressions:

```
^ $ \ . * + ? ( ) [ ] { } |
```

For example, the following regular expression matches the letter A followed by zero or more instances of the letter B (the asterisk metacharacter indicates this repetition), followed by the letter C:

```
/AB*C/
```

To include a metacharacter without its special meaning in a regular expression pattern, you must use the backslash (\) escape character. For example, the following regular expression matches the letter A followed by the letter B, followed by an asterisk, followed by the letter C:

```
var pattern:RegExp = /AB\*C/;
```

A *metasequence,* like a metacharacter, has special meaning in a regular expression. A metasequence is made up of more than one character. The following sections provide details on using metacharacters and metasequences.

About metacharacters

The following table summarizes the metacharacters that you can use in regular expressions:

Metacharacter	Description
`^` (caret)	Matches at the start of the string. With the `m` (`multiline`) flag set, the caret matches the start of a line as well (see "The m (multiline) flag" on page 300). Note that when used at the start of a character class, the caret indicates negation, not the start of a string. For more information, see"Character classes" on page 291.
`$` (dollar sign)	Matches at the end of the string. With the `m` (`multiline`) flag set, `$` matches the position before a newline (`\n`) character as well. For more information, see "The m (multiline) flag" on page 300.
`\` (backslash)	Escapes the special metacharacter meaning of special characters. Also, use the backslash character if you want to use a forward slash character in a regular expression literal, as in `/1\/2/` (to match the character 1, followed by the forward slash character, followed by the character 2).
`.` (dot)	Matches any single character. A dot matches a newline character (`\n`) only if the `s` (`dotall`) flag is set. For more information, see "The s (dotall) flag" on page 301.
`*` (star)	Matches the previous item repeated zero or more times. For more information, see "Quantifiers" on page 293.
`+` (plus)	Matches the previous item repeated one or more times. For more information, see "Quantifiers" on page 293.
`?` (question mark)	Matches the previous item repeated zero times or one time. For more information, see "Quantifiers" on page 293.
`(` and `)`	Defines groups within the regular expression. Use groups for the following: • To confine the scope of the | alternator: `/(a|b|c)d/` • To define the scope of a quantifier: `/(walla.){1,2}/` • In backreferences. For example, the `\1` in the following regular expression matches whatever matched the first parenthetical group of the pattern: `/(\w*) is repeated: \1/` For more information, see "Groups" on page 295.

Metacharacter	Description
`[` and `]`	Defines a character class, which defines possible matches for a single character: `/[aeiou]/` matches any one of the specified characters. Within character classes, use the hyphen (`-`) to designate a range of characters: `/[A-Z0-9]/` matches uppercase A through Z or 0 through 9. Within character classes, insert a backslash to escape the] and - characters: `/[+\-]\d+/` matches either `+` or `-` before one or more digits. Within character classes, other characters, which are normally metacharacters, are treated as normal characters (not metacharacters), without the need for a backslash: `/[$£]/` matches either `$ or £`. For more information, see "Character classes" on page 291.
`|` *(pipe)*	Used for alternation, to match either the part on the left side or the part on the right side: `/abc|xyz/` matches either `abc` or `xyz`.

About metasequences

Metasequences are sequences of characters that have special meaning in a regular expression pattern. The following table describes these metasequences:

Metasequence	Description

`{`*n*`}`
`{`*n*`,}`
and
`{`*n*`,`*n*`}`
	Specifies a numeric quantifier or quantifier range for the previous item:
	`/A{27}/` matches the character `A` repeated `27` times.
	`/A{3,}/` matches the character `A` repeated `3` or more times.
	`/A{3,5}/` matches the character `A` repeated `3` to `5` times.
	For more information, see "Quantifiers" on page 293.

`\b`	Matches at the position between a word character and a nonword character. If the first or last character in the string is a word character, also matches the start or end of the string.
`\B`	Matches at the position between two word characters. Also matches the position between two nonword characters.
`\d`	Matches a decimal digit.
`\D`	Matches any character other than a digit.
`\f`	Matches a form feed character.
`\n`	Matches the newline character.
`\r`	Matches the return character.

Metasequence	Description
`\s`	Matches any white-space character (a space, tab, newline, or return character).
`\S`	Matches any character other than a white-space character.
`\t`	Matches the tab character.
`\u`*`nnnn`*	Matches the Unicode character with the character code specified by the hexidecimal number *`nnnn`*. For example, `\u263a` is the smiley character.
`\v`	Matches a vertical feed character.
`\w`	Matches a word character (`A-Z`, `a-z`, `0-9`, or _). Note that `\w` does not match non-English characters, such as `é`, `ñ`, or `ç`.
`\W`	Matches any character other than a word character.
`\x`*`nn`*	Matches the character with the specified ASCII value, as defined by the hexidecimal number *`nn`*.

Character classes

You use character classes to specify a list of characters to match one position in the regular expression. You define character classes with square brackets (`[` and `]`). For example, the following regular expression defines a character class that matches `bag`, `beg`, `big`, `bog`, or `bug`:

```
/b[aeiou]g/
```

Escape sequences in character classes

Most metacharacters and metasequences that normally have special meanings in a regular expression *do not* have those same meanings inside a character class. For example, in a regular expression, the asterisk is used for repetition, but this is not the case when the asterisk appears in a character class. The following character class matches the asterisk literally, along with any of the other characters listed:

```
/[abc*123]/
```

However, the three characters listed in the following table do function as metacharacters, with special meaning, in character classes:

Metacharacter	Meaning in character classes
`]`	Defines the end of the character class.
`-`	Defines a range of characters (see "Ranges of characters in character classes" on page 292).
`\`	Defines metasequences and undoes the special meaning of metacharacters.

For any of these characters to be recognized as literal characters (without the special metacharacter meaning), you must precede the character with the backslash escape character. For example, the following regular expression includes a character class that matches any one of four symbols (`$`, `\`, `]`, or `-`):

```
/[$\\\]\-]/
```

In addition to the metacharacters that retain their special meanings, the following metasequences function as metasequences within character classes:

Metasequence	Meaning in character classes
`\n`	Matches a newline character.
`\r`	Matches a return character.
`\t`	Matches a tab character.
`\u`*nnnn*	Matches the character with the specified Unicode code point value (as defined by the hexidecimal number *nnnn*).
`\x`*nn*	Matches the character with the specified ASCII value (as defined by the hexidecimal number *nn*).

Other regular expression metasequences and metacharacters are treated as normal characters within a character class.

Ranges of characters in character classes

Use the hyphen to specify a range of characters, such as `A-Z`, `a-z`, or `0-9`. These characters must constitute a valid range in the character set. For example, the following character class matches any one of the characters in the range `a-z` or any digit:

```
/[a-z0-9]/
```

You can also use the `\xnn` ASCII character code to specify a range by ASCII value. For example, the following character class matches any character from a set of extended ASCII characters (such as `é` and `ê`):

```
/[\x80-\x9A]/
```

Negated character classes

When you use a caret (`^`) character at the beginning of a character class, it negates that class—any character not listed is considered a match. The following character class matches any character *except* for a lowercase letter (`a-z`) or a digit:

```
/[^a-z0-9]/
```

You must type the caret (`^`) character at the *beginning* of a character class to indicate negation. Otherwise, you are simply adding the caret character to the characters in the character class. For example, the following character class matches any one of a number of symbol characters, including the caret:

```
/[!.,#+*%$&^]/
```

Quantifiers

You use quantifiers to specify repetitions of characters or sequences in patterns, as follows:

Quantifier metacharacter	**Description**
`*` (star)	Matches the previous item repeated zero or more times.
`+` (plus)	Matches the previous item repeated one or more times.
`?` (question mark)	Matches the previous item repeated zero times or one time.

`{n}`
`{n,}`
and
`{n,n}`
	Specifies a numeric quantifier or quantifier range for the previous item:
	`/A{27}/` matches the character A repeated 27 times.
	`/A{3,}/` matches the character A repeated 3 or more times.
	`/A{3,5}/` matches the character A repeated 3 to 5 times.

You can apply a quantifier to a single character, to a character class, or to a group:

- `/a+/` matches the character `a` repeated one or more times.
- `/\d+/` matches one or more digits.
- `/[abc]+/` matches a repetition of one or more character, each of which is either `a`, `b`, or `c`.
- `/(very, )*/` matches the word `very` followed by a comma and a space repeated zero or more times.

You can use quantifiers within parenthetical groupings that have quantifiers applied to them. For example, the following quantifier matches strings such as `word` and `word-word-word`:

```
/\w+(-\w+)*/
```

By default, regular expressions perform what is known as *greedy matching.* Any subpattern in the regular expression (such as `.*`) tries to match as many characters in the string as possible before moving forward to the next part of the regular expression. For example, consider the following regular expression and string:

```
var pattern:RegExp = /<p>.*<\/p>/;
str:String = "<p>Paragraph 1</p> <p>Paragraph 2</p>";
```

The regular expression matches the entire string:

```
<p>Paragraph 1</p> <p>Paragraph 2</p>
```

Suppose, however, that you want to match only one `<p>...</p>` grouping. You can do this with the following:

```
<p>Paragraph 1</p>
```

Add a question mark (`?`) after any quantifier to change it to what is known as a *lazy quantifier*. For example, the following regular expression, which uses the lazy `*?` quantifier, matches `<p>` followed by the minimum number of characters possible (lazy), followed by `</p>`:

```
/<p>.*?<\/p>/
```

Keep in mind the following points about quantifiers:

- The quantifiers `{0}` and `{0,0}` do not exclude an item from a match.
- Do not combine multiple quantifiers, as in `/abc+*/`.
- The dot (.) does not span lines unless the `s` (`dotall`) flag is set, even if it is followed by a `*` quantifier. For example, consider the following code:

```
var str:String = "<p>Test\n";
str += "Multiline</p>";
var re:RegExp = /<p>.*<\/p>/;
trace(str.match(re)); // null;

re = /<p>.*<\/p>/s;
trace(str.match(re));
  // output: <p>Test
  //          Multiline</p>
```

For more information, see "The s (dotall) flag" on page 301.

Alternation

Use the `|` (bar) character in a regular expression to have the regular expression engine consider alternatives for a match. For example, the following regular expression matches any one of the words `cat`, `dog`, `pig`, `rat`:

```
var pattern:RegExp = /cat|dog|pig|rat/;
```

You can use parentheses to define groups to restrict the scope of the | alternator. The following regular expression matches `cat` followed by `nap` or `nip`:

```
var pattern:RegExp = /cat(nap|nip)/;
```

For more information, see "Groups" on page 295.

The following two regular expressions, one using the `|` alternator, the other using a character class (defined with `[` and `]`), are equivalent:

```
/1|3|5|7|9/
/[13579]/
```

For more information, see "Character classes" on page 291.

Groups

You can specify a group in a regular expression by using parentheses, as follows:

```
/class-(\d*)/
```

A group is a subsection of a pattern. You can use groups to do the following things:

- Apply a quantifier to more than one character.
- Delineate subpatterns to be applied with alternation (by using the `|` character).
- Capture substring matches (for example, by using `\1` in a regular expression to match a previously matched group, or by using `$1` similarly in the `replace()` method of the String class).

The following sections provide details on these uses of groups.

Using groups with quantifiers

If you do not use a group, a quantifier applies to the character or character class that precedes it, as the following shows:

```
var pattern:RegExp = /ab*/ ;
// matches the character a followed by
// zero or more occurrences of the character b

pattern = /a\d+/;
// matches the character a followed by
// one or more digits

pattern = /a[123]{1,3}/;
// matches the character a followed by
// one to three occurrences of either 1, 2, or 3
```

However, you can use a group to apply a quantifier to more than one character or character class:

```
var pattern:RegExp = /(ab)*/;
// matches zero or more occurrences of the character a
// followed by the character b, such as ababab

pattern = /(a\d)+/;
// matches one or more occurrences of the character a followed by
// a digit, such as a1a5a8a3

pattern = /(spam ){1,3}/;
// matches 1 to 3 occurrences of the word spam followed by a space
```

For more information on quantifiers, see "Quantifiers" on page 293.

Using groups with the alternator (|) character

You can use groups to define the group of characters to which you want to apply an alternator (|) character, as follows:

```
var pattern:RegExp = /cat|dog/;
// matches cat or dog

pattern = /ca(t|d)og/;
// matches catog or cadog
```

Using groups to capture substring matches

When you define a standard parenthetical group in a pattern, you can later refer to it in the regular expression. This is known as a *backreference*, and these sorts of groups are known as *capturing groups*. For example, in the following regular expression, the sequence `\1` matches whatever substring matched the capturing parenthetical group:

```
var pattern:RegExp = /(\d+)-by-\1/;
// matches the following: 48-by-48
```

You can specify up to 99 of these backreferences in a regular expression by typing `\1`, `\2`, ... , `\99`.

Similarly, in the `replace()` method of the String class, you can use `$1-$99` to insert captured group substring matches in the replacement string:

```
var pattern:RegExp = /Hi, (\w+)\./;
var str:String = "Hi, Bob.";
trace(str.replace(pattern, "$1, hello."));
// output: Bob, hello.
```

Also, if you use capturing groups, the `exec()` method of the RegExp class and the `match()` method of the String class return substrings that match the capturing groups:

```
var pattern:RegExp = /(\w+)@(\w+).(\w+)/;
var str:String = "bob@example.com";
trace(pattern.exec(str));
   // bob@test.com,bob,example,com
```

Using noncapturing groups and lookahead groups

A noncapturing group is one that is used for grouping only; it is not "collected," and it does not match numbered backreferences. Use `(?:` and `)` to define noncapturing groups, as follows:

```
var pattern = /(?:com|org|net);
```

For example, note the difference between putting `(com|org)` in a capturing versus a noncapturing group (the `exec()` method lists capturing groups after the complete match):

```
var pattern:RegExp = /(\w+)@(\w+).(com|org)/;
var str:String = "bob@example.com";
trace(pattern.exec(str));
  // bob@test.com,bob,example,com

//noncapturing:
var pattern:RegExp = /(\w+)@(\w+).(?:com|org)/;
var str:String = "bob@example.com";
trace(pattern.exec(str));
  // bob@test.com,bob,example
```

A special type of noncapturing group is the *lookahead group,* of which there are two types: the *positive lookahead group* and the *negative lookahead group.*

Use `(?=` and `)` to define a positive lookahead group, which specifies that the subpattern in the group must match at the position. However, the portion of the string that matches the positive lookahead group can match remaining patterns in the regular expression. For example, because `(?=e)` is a positive lookahead group in the following code, the character `e` that it matches can be matched by a subsequent part of the regular expression—in this case, the capturing group, `\w*)`:

```
var pattern:RegExp = /sh(?=e)(\w*)/i;
var str:String = "Shelly sells seashells by the seashore";
trace(pattern.exec(str));
// Shelly,elly
```

Use `(?!` and `)` to define a negative lookahead group that specifies that the subpattern in the group must *not* match at the position. For example:

```
var pattern:RegExp = /sh(?!e)(\w*)/i;
var str:String = "She sells seashells by the seashore";
trace(pattern.exec(str));
// shore,ore
```

Using named groups

A named group is a type of group in a regular expression that is given a named identifier. Use `(?P<name>` and `)` to define the named group. For example, the following regular expression includes a named group with the identifier named `digits`:

```
var pattern = /[a-z]+(?P<digits>\d+)[a-z]+/;
```

When you use the `exec()` method, a matching named group is added as a property of the `result` array:

```
var myPattern:RegExp = /([a-z]+)(?P<digits>\d+)[a-z]+/;
var str:String = "a123bcd";
var result:Array = myPattern.exec(str);
trace(result.digits); // 123
```

Here is another example, which uses two named groups, with the identifiers `name` and `dom`:

```
var emailPattern:RegExp =
  /(?P<name>(\w|[_.\-])+)@(?P<dom>((\w|-)+))+\.\w{2,4}+/;
var address:String = "bob@example.com";
var result:Array = emailPattern.exec(address);
trace(result.name); // bob
trace(result.dom); // example
```

NOTE

Named groups are not part of the ECMAScript language specification. They are an added feature in ActionScript 3.0.

Flags and properties

The following table lists the five flags that you can set for regular expressions. Each flag can be accessed as a property of the regular expression object.

Flag	Property	Description
`g`	`global`	Matches more than one match.
`i`	`ignoreCase`	Case-insensitive matching. Applies to the `A`—`Z` and `a`—`z` characters, but not to extended characters such as `É` and `é`.
`m`	`multiline`	With this flag set, `$` and `^` can match the beginning of a line and end of a line, respectively.
`s`	`dotall`	With this flag set, `.` (dot) can match the newline character (`\n`).
`x`	`extended`	Allows extended regular expressions. You can type spaces in the regular expression, which are ignored as part of the pattern. This lets you type regular expression code more legibly.

Note that these properties are read-only. You can set the flags (`g`, `i`, `m`, `s`, `x`) when you set a regular expression variable, as follows:

```
var re:RegExp = /abc/gimsx;
```

However, you cannot directly set the named properties. For instance, the following code results in an error:

```
var re:RegExp = /abc/;
re.global = true; // This generates an error.
```

By default, unless you specify them in the regular expression declaration, the flags are not set, and the corresponding properties are also set to `false`.

Additionally, there are two other properties of a regular expression:

- The `lastIndex` property specifies the index position in the string to use for the next call to the `exec()` or `test()` method of a regular expression.
- The `source` property specifies the string that defines the pattern portion of the regular expression.

The g (global) flag

When the `g` (`global`) flag is *not* included, a regular expression matches no more than one match. For example, with the `g` flag not included in the regular expression, the `String.match()` method returns only one matching substring:

```
var str:String = "she sells seashells by the seashore.";
var pattern:RegExp = /sh\w*/;
trace(str.match(pattern)) // output: she
```

When the `g` flag is set, the `Sting.match()` method returns multiple matches, as follows:

```
var str:String = "she sells seashells by the seashore.";
var pattern:RegExp = /sh\w*/g;
// The same pattern, but this time the g flag IS set.
trace(str.match(pattern)); // output: she,shells,shore
```

The i (ignoreCase) flag

By default, regular expression matches are case-sensitive. When you set the `i` (`ignoreCase`) flag, case sensitivity is ignored. For example, the lowercase `s` in the regular expression does not match the uppercase letter `S`, the first character of the string:

```
var str:String = "She sells seashells by the seashore.";
trace(str.search(/sh/)); // output: 13 -- Not the first character
```

With the `i` flag set, however, the regular expression does match the capital letter `S`:

```
var str:String = "She sells seashells by the seashore.";
trace(str.search(/sh/i)); // output: 0
```

The `i` flag ignores case sensitivity only for the `A`–`Z` and `a`–`z` characters, but not for extended characters such as `É` and `é`.

The m (multiline) flag

If the `m` (`multiline`) flag is not set, the `^` matches the beginning of the string and the `$` matches the end of the string. If the `m` flag is set, these characters match the beginning of a line and end of a line, respectively. Consider the following string, which includes a newline character:

```
var str:String = "Test\n";
str += "Multiline";
trace(str.match(/^\w*/g)); // Match a word at the beginning of the string.
```

Even though the `g` (`global`) flag is set in the regular expression, the `match()` method matches only one substring, since there is only one match for the `^`—the beginning of the string. The output is:

```
Test
```

Here is the same code with the `m` flag set:

```
var str:String = "Test\n";
str += "Multiline";
trace(str.match(/^\w*/gm)); // Match a word at the beginning of lines.
```

This time, the output includes the words at the beginning of both lines:

```
Test,Multiline
```

Note that only the `\n` character signals the end of a line. The following characters do not:

- Return (`\r`) character
- Unicode line-separator (`\u2028`) character
- Unicode paragraph-separator (`\u2029`) character

The s (dotall) flag

If the `s` (`dotall` or "dot all") flag is not set, a dot (`.`) in a regular expression pattern does not match a newline character (`\n`). So for the following example, there is no match:

```
var str:String = "<p>Test\n";
str += "Multiline</p>";
var re:RegExp = /<p>.*?<\/p>/;
trace(str.match(re));
```

However, if the `s` flag is set, the dot matches the newline character:

```
var str:String = "<p>Test\n";
str += "Multiline</p>";
var re:RegExp = /<p>.*?<\/p>/s;
trace(str.match(re));
```

In this case, the match is the entire substring within the `<p>` tags, including the newline character:

```
<p>Test
Multiline</p>
```

The x (extended) flag

Regular expressions can be difficult to read, especially when they include a lot of metasymbols and metasequences. For example:

```
/<p(>|(\s*[^>]*>)).*?<\/p>/gi
```

When you use the `x` (`extended`) flag in a regular expression, any blank spaces that you type in the pattern are ignored. For example, the following regular expression is identical to the previous example:

```
/     <p    (>  |  (\s*  [^>]*  >))    .*?    <\/p>  /gix
```

If you have the `x` flag set and do want to match a blank space character, precede the blank space with a backslash. For example, the following two regular expressions are equivalent:

```
/foo bar/
/foo \ bar/x
```

The lastIndex property

The `lastIndex` property specifies the index position in the string at which to start the next search. This property affects the `exec()` and `test()` methods called on a regular expression that has the `g` flag set to `true`. For example, consider the following code:

```
var pattern:RegExp = /p\w*/gi;
var str:String = "Pedro Piper picked a peck of pickled peppers.";
trace(pattern.lastIndex);
var result:Object = pattern.exec(str);
while (result != null)
{
  trace(pattern.lastIndex);
  result = pattern.exec(str);
}
```

The `lastIndex` property is set to 0 by default (to start searches at the beginning of the string). After each match, it is set to the index position following the match. Therefore, the output for the preceding code is the following:

```
0
5
11
18
25
36
44
```

If the `global` flag is set to `false`, the `exec()` and `test()` methods do not use or set the `lastIndex` property.

The `match()`, `replace()`, and `search()` methods of the String class start all searches from the beginning of the string, regardless of the setting of the `lastIndex` property of the regular expression used in the call to the method. (However, the `match()` method does set `lastIndex` to 0.)

You can set the `lastIndex` property to adjust the starting position in the string for regular expression matching.

The source property

The `source` property specifies the string that defines the pattern portion of a regular expression. For example:

```
var pattern:RegExp = /foo/gi;
trace(pattern.source); // foo
```

Methods for using regular expressions with strings

The RegExp class includes two methods: `exec()` and `test()`.

In addition to the `exec()` and `test()` methods of the RegExp class, the String class includes the following methods that let you match regular expressions in strings: `match()`, `replace()`, `search()`, and `splice()`.

The test() method

The `test()` method of the RegExp class simply checks the supplied string to see if it contains a match for the regular expression, as the following example shows:

```
var pattern:RegExp = /Class-\w/;
var str = "Class-A";
trace(pattern.test(str)); // output: true
```

The exec() method

The `exec()` method of the RegExp class checks the supplied string for a match of the regular expression and returns an array with the following:

- The matching substring
- Substring matches for any parenthetical groups in the regular expression

The array also includes an `index` property, indicating the index position of the start of the substring match.

For example, consider the following code:

```
var pattern:RegExp = /\d{3}\-\d{3}-\d{4}/; //U.S phone number
var str:String = "phone: 415-555-1212";
var result:Array = pattern.exec(str);
trace(result.index, " - ", result);
// 7  -  415-555-1212
```

Use the `exec()` method multiple times to match multiple substrings when the `g` (`global`) flag is set for the regular expression:

```
var pattern:RegExp = /\w*sh\w*/gi;
var str:String = "She sells seashells by the seashore";
var result:Array = pattern.exec(str);

while (result != null)
{
    trace(result.index, "\t", pattern.lastIndex, "\t", result);
    result = pattern.exec(str);
}
//output:
// 0  3  She
// 10  19  seashells
// 27  35  seashore
```

String methods that use RegExp parameters

The following methods of the String class take regular expressions as parameters: `match()`, `replace()`, `search()`, and `split()`. For more information on these methods, see "Finding patterns in strings and replacing substrings" on page 216.

Example: A Wiki parser

This simple Wiki text conversion example illustrates a number of uses for regular expressions:

- Converting lines of text that match a source Wiki pattern to the appropriate HTML output strings.
- Using a regular expression to convert URL patterns to HTML `<a>` hyperlink tags.
- Using a regular expression to convert U.S. dollar strings (such as `"$9.95"`) to euro strings (such as `"8.24 €"`).

The WikiEditor application files can be found in the folder Samples/WikiEditor. The application consists of the following files:

File	Description
WikiEditor.mxml	The main application file in MXML for Flex.
com/example/programmingas3/regExpExamples/WikiParser.as	A class that includes methods that use regular expressions to convert Wiki input text patterns to the equivalent HTML output.
com/example/programmingas3/regExpExamples/URLParser.as	A class that includes methods that use regular expressions to convert URL strings to HTML `<a>` hyperlink tags.
com/example/programmingas3/regExpExamples/CurrencyConverter.as	A class that includes methods that use regular expressions to convert U.S. dollar strings to euro strings.

Defining the WikiParser class

The WikiParser class includes methods that convert Wiki input text into the equivalent HTML output. This is not a very robust Wiki conversion application, but it does illustrate some good uses of regular expressions for pattern matching and string conversion.

The constructor function, along with the `setWikiData()` method, simply initializes a sample string of Wiki input text, as follows:

```
public function WikiParser()
{
  wikiData = setWikiData();
}
```

When the user clicks the Test button in the sample application, the application invokes the `parseWikiString()` method of the WikiParser object. This method calls a number of other methods, which in turn assemble the resulting HTML string.

```
public function parseWikiString(wikiString:String):String
{
  var result:String = parseBold(wikiString);
  result = parseItalic(result);
  result = linesToParagraphs(result);
  result = parseBullets(result);
  return result;
}
```

Each of the methods called—`parseBold()`, `parseItalic()`, `linesToParagraphs()`, and `parseBullets()`—uses the `replace()` method of the string to replace matching patterns, defined by a regular expression, in order to transform the input Wiki text into HTML-formatted text.

Converting boldface and italic patterns

The `parseBold()` method looks for a Wiki boldface text pattern (such as `'''foo'''`) and transforms it into its HTML equivalent (such as `<b>foo</b>`), as follows:

```
private function parseBold(input:String):String
{
  var pattern:RegExp = /'''(.*?)'''/g;
  return input.replace(pattern, "<b>$1</b>");
}
```

Note that the `(.?*)` portion of the regular expression matches any number of characters (`*`) between the two defining `'''` patterns. The `?` quantifier makes the match nongreedy, so that for a string such as `'''aaa''' bbb '''ccc'''`, the first matched string will be `'''aaa'''` and not the entire string (which starts and ends with the `'''` pattern).

The parentheses in the regular expression define a capturing group, and the `replace()` method refers to this group by using the `$1` code in the replacement string. The `g` (`global`) flag in the regular expression ensures that the `replace()` method replaces all matches in the string (not simply the first one).

The `parseItalic()` method works similarly to the `parseBold()` method, except that it checks for two apostrophes (`''`) as the delimiter for italic text (not three):

```
private function parseItalic(input:String):String
{
  var pattern:RegExp = /''(.*?)''/g;
  return input.replace(pattern, "<i>$1</i>");
}
```

Converting bullet patterns

As the following example shows, the `parseBullet()` method looks for the Wiki bullet line pattern (such as `* foo`) and transforms it into its HTML equivalent (such as `<li>foo</li>`):

```
private function parseBullets(input:String):String
{
  var pattern:RegExp = /^\*(.*)/gm;
  return input.replace(pattern, "<li>$1</li>");
}
```

The `^` symbol at the beginning of the regular expression matches the beginning of a line. The `m` (`multiline`) flag in the regular expression causes the regular expression to match the `^` symbol against the start of a line, not simply the start of the string.

The `\*` pattern matches an asterisk character (the backslash is used to signal a literal asterisk instead of a `*` quantifier).

The parentheses in the regular expression define a capturing group, and the `replace()` method refers to this group by using the `$1` code in the replacement string. The `g` (`global`) flag in the regular expression ensures that the `replace()` method replaces all matches in the string (not simply the first one).

Converting paragraph Wiki patterns

The `linesToParagraphs()` method converts each line in the input Wiki string to an HTML `<p>` paragraph tag. These lines in the method strip out empty lines from the input Wiki string:

```
var pattern:RegExp = /^$/gm;
var result:String = input.replace(pattern, "");
```

The ^ and $ symbols the regular expression match the beginning and end of a line. The m (multiline) flag in the regular expression causes the regular expression to match the ^ symbol against the start of a line, not simply the start of the string.

The replace() method replaces all matching substrings (empty lines) with an empty string (""). The g (global) flag in the regular expression ensures that the replace() method replaces all matches in the string (not simply the first one).

Converting URLs to HTML <a> tags

When the user clicks the Test button in the sample application, if the user selected the urlToATag check box, the application calls the URLParser.urlToATag() static method to convert URL strings from the input Wiki string into HTML <a> tags.

```
var protocol:String = "((?:http|ftp)://)";
var urlPart:String = "([a-z0-9_-]+\.[a-z0-9_-]+)";
var optionalUrlPart:String = "(\.[a-z0-9_-]*)";
var urlPattern:RegExp = new RegExp(protocol + urlPart + optionalUrlPart,
                        "ig");
var result:String = input.replace(urlPattern,
                "<a href='$1$2$3'><u>$1$2$3</u></a>");
```

The RegExp() constructor function is used to assemble a regular expression (urlPattern) from a number of constituent parts. These constituent parts are each strings that define part of the regular expression pattern.

The first part of the regular expression pattern, defined by the protocol string, defines an URL protocol: either http:// or ftp://. The parentheses define a noncapturing group, indicated by the ? symbol. This means that the parentheses are simply used to define a group for the | alternation pattern; the group will not match backreference codes ($1, $2, $3) in the replacement string of the replace() method.

The other constituent parts of the regular expression each use capturing groups (indicated by parentheses in the pattern), which are then used in the backreference codes ($1, $2, $3) in the replacement string of the replace() method.

The part of the pattern defined by the urlPart string matches *at least* one of the following characters: a-z, 0-9, _, or -. The + quantifier indicates that at least one character is matched. The \. indicates a required dot (.) character. And the remainder matches another string of at least one of these characters: a-z, 0-9, _, or -.

The part of the pattern defined by the optionalUrlPart string matches *zero or more* of the following: a dot (.) character followed by any number of alphanumeric characters (including _ and -). The * quantifier indicates that zero or more characters are matched.

The call to the `replace()` method employs the regular expression and assembles the replacement HTML string, using backreferences.

The `urlToATag()` method then calls the `emailToATag()` method, which uses similar techniques to replace e-mail patterns with HTML `<a>` hyperlink strings. The regular expressions used to match HTTP, FTP, and e-mail URLs in this sample file are fairly simple, for the purposes of exemplification; there are much more complicated regular expressions for matching such URLs more correctly.

Converting U.S. dollar strings to euro strings

When the user clicks the Test button in the sample application, if the user selected the `dollarToEuro` check box, the application calls the `CurrencyConverter.usdToEuro()` static method to convert U.S. dollar strings (such as `"$9.95"`) to euro strings (such as `"8.24 €"`), as follows:

```
var usdPrice:RegExp = /\$([\d,]+.\d+)+/g;
return input.replace(usdPrice, usdStrToEuroStr);
```

The first line defines a simple pattern for matching U.S. dollar strings. Notice that the `$` character is preceded with the backslash (`\`) escape character.

The `replace()` method uses the regular expression as the pattern matcher, and it calls the `usdStrToEuroStr()` function to determine the replacement string (a value in euros).

When a function name is used as the second parameter of the `replace()` method, the following are passed as parameters to the called function:

- The matching portion of the string.
- Any captured parenthetical group matches. The number of arguments passed this way varies depending on the number of captured parenthetical group matches. You can determine the number of captured parenthetical group matches by checking `arguments.length - 3` within the function code.
- The index position in the string where the match begins.
- The complete string.

The `usdStrToEuroStr()` method converts U.S. dollar string patterns to euro strings, as follows:

```
private function usdToEuro(...args):String
{
  var usd:String = args[1];
  usd = usd.replace(",", "");
  var exchangeRate:Number = 0.828017;
  var euro:Number = Number(usd) * exchangeRate;
  trace(usd, Number(usd), euro);
```

```
    const euroSymbol:String = String.fromCharCode(8364); // €
    return euro.toFixed(2) + " " + euroSymbol;
}
```

Note that `args[1]` represents the captured parenthetical group matched by the `usdPrice` regular expression. This is the numerical portion of the U.S. dollar string: that is, the dollar amount without the `$` sign. The method applies an exchange rate conversion and returns the resulting string (with a trailing `€` symbol instead of a leading $ symbol).

CHAPTER 11

Working with XML

11

ActionScript 3.0 includes a group of classes based on the ECMAScript for XML (E4X) specification (ECMA-357 edition 2). These classes include powerful and easy-to-use functionality for working with XML data. Using E4X, you will be able to develop code with XML data faster than was possible with previous programming techniques. As an added benefit, the code you produce will be easier to read.

This chapter describes how to use E4X to process XML data.

Contents

A quick introduction to XML

This chapter assumes that you are familiar with basic XML concepts. However, even if you are new to XML, you may be able to get started working with basic XML methods by using the information in this chapter. As a basic introduction, consider the following XML document:

```
<order xmlns = "http://www.example.com/xml">
  <book ISBN="0942407296">
    <title>Baking Extravagant Pastries with Kumquats</title>
    <author>
      <lastName>Contino</lastName>
      <firstName>Chuck</firstName>
    </author>
    <pageCount>238</pageCount>
  </book>
  <book ISBN="0865436401">
    <title>Emu Care and Breeding</title>
    <editor>
      <lastName>Case</lastName>
      <firstName>Justin</firstName>
    </editor>
    <pageCount>115</pageCount>
  </book>
</order>
```

This document contains two `book` *elements* (also known as *nodes*). The first `book` element has three *child elements*, with the *names* `title`, `author`, and `pageCount`; the `author` element has two child elements. The first `book` element has an `ISBN` *attribute* (with the value `"0942407296"`). The *content* of the first `book` element is its collection of child elements. The content of the `firstName` element is the text `"Chuck"`. The entire XML document has a default *namespace*, defined at the fictitious URL http://www.example.com/xml. This namespace defines the schema for this type of XML document.

ActionScript 3.0 includes new operators—such as the dot (`.`) and attribute identifier (`@`) operators in the following example—for working with XML data. If you assign the previous sample XML data to an object named `order1`, the following statements are valid code:

```
trace(order1.book[0].author.firstName); // Chuck
trace(order1.book.(@ISBN=="0865436401").pageCount; // 115
delete order1.book[0];
```

The new operators and other new E4X classes, methods, and properties are discussed in this chapter.

Since many server-side applications use XML to structure data, you can use the XML classes in ActionScript to create sophisticated rich Internet applications, such as those that connect to web services. A web service is a means to connect applications—for example, an Adobe Flash Player 9 application and an application on a web server)—through a common standard such as the Simple Object Access Protocol (SOAP).

The E4X approach to XML processing

The ECMAScript for XML specification defines a set of classes and functionality for working with XML data. These classes and functionality are known collectively as E4X. ActionScript 3.0 includes the following E4X classes: XML, XMLList, QName, and Namespace.

The methods, properties, and operators of the E4X classes are designed with the following goals:

- Simplicity—Where possible, E4X makes it easier to write and understand code for working with XML data.
- Consistency—The methods and reasoning behind E4X are internally consistent and consistent with other parts of ActionScript.
- Familiarity—You manipulate XML data with well-known operators, such as the dot (.) operator.

NOTE

There was an XML class in ActionScript 2.0. In ActionScript 3.0 it has been renamed XMLDocument, so that it does not conflict with the ActionScript 3.0 XML class that is part of E4X. In ActionScript 3.0, the legacy classes–XMLDocument, XMLNode, XMLParser, and XMLTag–are included in the flash.xml package primarily for legacy support. The new E4X classes are core classes; you need not import a package to use them. This chapter does not go into detail on the legacy ActionScript 2.0 XML classes. For details on these, see the flash.xml package in the *ActionScript 3.0 Language Reference*.

Here is an example of manipulating data with E4X:

```
var myXML:XML =
  <order>
    <item id='1'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
    <item id='2'>
      <menuName>fries</menuName>
      <price>1.45</price>
    </item>
  </order>
```

Often, your application will load XML data from an external source, such as a web service or a RSS feed. However, for clarity, the examples in this chapter assign XML data as literals.

As the following code shows, E4X includes some intuitive operators, such as the dot (`.`) and attribute identifier (`@`) operators, for accessing properties and attributes in the XML:

```
trace(myXML.item[0].menuName); // Output: burger
trace(myXML.item.(@id==2).menuName); // Output: fries
trace(myXML.item.(menuName=="burger").price); // Output: 3.95
```

Use the `appendChild()` method to assign a new child node to the XML, as the following snippet shows:

```
var newItem:XML =
  <item id="3">
    <menuName>medium cola</menuName>
    <price>1.25</price>
  </item>

myXML.appendChild(newItem);
```

Use the `@` and `.` operators not only to read data, but also to assign data, as in the following:

```
myXML.item[0].menuName="regular burger";
myXML.item[1].menuName="small fries";
myXML.item[2].menuName="medium cola";

myXML.item.(menuName=="regular burger").@quantity = "2";
myXML.item.(menuName=="small fries").@quantity = "2";
myXML.item.(menuName=="medium cola").@quantity = "2";
```

Use a `for` loop to iterate through nodes of the XML, as follows:

```
var total:Number = 0;
for each (var property:XML in myXML.item)
{
  var q:int = Number(property.@quantity);
  var p:Number = Number(property.price);
  var itemTotal:Number = q * p;
  total += itemTotal;
  trace(q + " " + property.menuName + " $" + itemTotal.toFixed(2))
}
trace("Total: $", total.toFixed(2));
```

XML objects

An XML object may represent an XML element, attribute, comment, processing instruction, or text element.

An XML object is classified as having either *simple content* or *complex content*. An XML object that has child nodes is classified as having complex content. An XML object is said to have simple content if it is any one of the following: an attribute, a comment, a processing instruction, or a text node.

For example, the following XML object contains complex content, including a comment and a processing instruction:

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;
var x1:XML =
  <order>
    <!--This is a comment. -->
    <?PROC_INSTR sample ?>
    <item id='1'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
    <item id='2'>
      <menuName>fries</menuName>
      <price>1.45</price>
    </item>
  </order>
```

As the following example shows, you can now use the `comments()` and `processingInstructions()` methods to create new XML objects, a comment and a processing instruction:

```
var x2:XML = x1.comments()[0];
var x3:XML = x1.processingInstructions()[0];
```

XML properties

The XML class has five static properties:

- The `ignoreComments` and `ignoreProcessingInstructions` properties determine whether comments or processing instructions are ignored when the XML object is parsed.
- The `ignoreWhitespace` property determines whether white space characters are ignored in element tags and embedded expressions that are separated only by white space characters.
- The `prettyIndent` and `prettyPrinting` properties are used to format the text that is returned by the `toString()` and `toXMLString()` methods of the XML class.

For details on these properties, see the *ActionScript 3.0 Language Reference*.

XML methods

The following methods allow you to work with the hierarchical structure of XML objects:

- `appendChild()`
- `child()`
- `childIndex()`
- `children()`
- `descendants()`
- `elements()`
- `insertChildAfter()`
- `insertChildBefore()`
- `parent()`
- `prependChild()`

The following methods allow you to work with XML object attributes:

- `attribute()`
- `attributes()`

The following methods allow you to you work with XML object properties:

- `hasOwnProperty()`
- `propertyIsEnumerable()`
- `replace()`
- `setChildren()`

The following methods are for working with qualified names and namespaces:

- `addNamespace()`
- `inScopeNamespaces()`
- `localName()`
- `name()`
- `namespace()`
- `namespaceDeclarations()`
- `removeNamespace()`
- `setLocalName()`
- `setName()`
- `setNamespace()`

The following methods are for working with and determining certain types of XML content:

- `comments()`
- `hasComplexContent()`
- `hasSimpleContent()`
- `nodeKind()`
- `processingInstructions()`
- `text()`

The following methods are for conversion to strings and for formatting XML objects:

- `defaultSettings()`
- `setSettings()`
- `settings()`
- `normalize()`
- `toString()`
- `toXMLString()`

There are a few additional methods:

- `contains()`
- `copy()`
- `valueOf()`
- `length()`

For details on these methods, see the *ActionScript 3.0 Language Reference*.

XMLList objects

An XMLList instance represents an arbitrary collection of XML objects. It can contain full XML documents, XML fragments, or the results of an XML query.

The following methods allow you to work with the hierarchical structure of XMLList objects:

- `child()`
- `children()`
- `descendants()`
- `elements()`
- `parent()`

The following methods allow you to work with XMLList object attributes:

- `attribute()`
- `attributes()`

The following methods allow you to you work with XMLList properties:

- `hasOwnProperty()`
- `propertyIsEnumerable()`

The following methods are for working with and determining certain types of XML content:

- `comments()`
- `hasComplexContent()`
- `hasSimpleContent()`
- `processingInstructions()`
- `text()`

The following are for conversion to strings and for formatting the XMLList object:

- `normalize()`
- `toString()`
- `toXMLString()`

There are a few additional methods:

- `contains()`
- `copy()`
- `length()`
- `valueOf()`

For details on these methods, see the *ActionScript 3.0 Language Reference*.

For an XMLList object that contains exactly one XML element, you can use all properties and methods of the XML class, because an XMLList with one XML element is treated the same as an XML object. For example, in the following code, because `doc.div` is an XMLList object containing one element, you can use the `appendChild()` method from the XML class:

```
var doc:XML =
    <body>
      <div>
        <p>Hello</p>
      </div>
    </body>;
doc.div.appendChild(<p>World</p>);
```

For a list of XML properties and methods, see "XML objects" on page 315.

Initializing XML variables

You can assign an XML literal to an XML object, as follows:

```
var myXML:XML =
  <order>
    <item id='1'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
    <item id='2'>
      <menuName>fries</menuName>
      <price>1.45</price>
    </item>
  </order>
```

As the following snippet shows, you can also use the `new` constructor to create an instance of an XML object from a string that contains XML data:

```
var str:String = "<order><item id='1'><menuName>burger</menuName>"
                + "<price>3.95</price></item></order>";
var myXML:XML = new XML(str);
```

If the XML data in the string is not well formed (for example, if a closing tag is missing), you will see a run-time error.

You can also pass data by reference (from other variables) into an XML object, as the following example shows:

```
var tagname:String = "item";
var attributename:String = "id";
var attributevalue:String = “5”;
var content:String = "Chicken";
var x:XML = <{tagname} {attributename}={attributevalue}>{content}</
  {tagname}>;
trace(x.toXMLString())
  // Output: <item id="5">Chicken</item>
```

To load XML data from a URL, use the URLLoader class, as the following example shows:

```
import flash.events.Event;
import flash.net.URLLoader;
import flash.net.URLRequest;

var externalXML:XML;
var loader:URLLoader = new URLLoader();
var request:URLRequest = new URLRequest("xmlFile.xml");
loader.load(request);
loader.addEventListener(Event.COMPLETE, onComplete);

function onComplete(event:Event):void
{
    var loader:URLLoader = event.target as URLLoader;
    if (loader != null)
    {
        externalXML = new XML(loader.data);
        trace(externalXML.toXMLString());
    }
    else
    {
        trace("loader is not a URLLoader!");
    }
}
```

To read XML data from a socket connection, use the XMLSocket class. For more information, see the XMLSocket entry in the *ActionScript 3.0 Language Reference*.

Assembling and transforming XML objects

Use the `prependChild()` method or the `appendChild()` method to add a property to the beginning or end of an XML object's list of properties, as the following example shows:

```
var x1:XML = <p>Line 1</p>
var x2:XML = <p>Line 2</p>
var x:XML = <body></body>
x = x.appendChild(x1);
x = x.appendChild(x2);
x = x.prependChild(<p>Line 0</p>);
    // x == <body><p>Line 0</p><p>Line 1</p><p>Line 2</p></body>
```

Use the `insertChildBefore()` method or the `insertChildAfter()` method to add a property before or after a specified property, as follows:

```
var x:XML =
    <body>
        <p>Paragraph 1</p>
        <p>Paragraph 2</p>
    </body>
var newNode:XML = <p>Paragraph 1.5</p>
x = x.insertChildAfter(x.p[0], newNode)
x = x.insertChildBefore(x.p[2], <p>Paragraph 1.75</p>)
```

As the following example shows, you can also use curly brace operators ({ and }) to pass data by reference (from other variables) when constructing XML objects:

```
var ids:Array = [121, 122, 123];
var names:Array = [["Murphy","Pat"], ["Thibaut","Jean"], ["Smith","Vijay"]]
var x:XML = new XML("<employeeList></employeeList>");

for (var i:int = 0; i < 3; i++)
{
  var newnode:XML = new XML();
  newnode =
    <employee id={ids[i]}>
      <last>{names[i][0]}</last>
      <first>{names[i][1]}</first>
    </employee>;

  x = x.appendChild(newnode)
}
```

You can assign properties and attributes to an XML object by using the = operator, as in the following:

```
var x:XML =
  <employee>
    <lastname>Smith</lastname>
  </employee>
x.firstname = "Jean";
x.@id = "239";
```

This sets the XML object `x` to the following:

```
<employee id="239">
  <lastname>Smith</lastname>
  <firstname>Jean</firstname>
</employee>
```

You can use the + and += operators to concatenate XMLList objects:

```
var x1:XML = <a>test1</a>
var x2:XML = <b>test2</b>
var xList:XMLList = x1 + x2;
xList += <c>test3</c>
```

This sets the XMLList object `xList` to the following:

```
<a>test1</a>
<b>test2</b>
<c>test3</c>
```

Traversing XML structures

One of the powerful features of XML is its ability to provide complex, nested data via a linear string of text characters. When you load data into an XML object, ActionScript parses the data and loads its hierarchical structure into memory (or it sends a run-time error if the XML data is not well formed).

The operators and methods of the XML and XMLList objects make it easy to traverse the structure of XML data.

Use the dot (.) operator and the descendent accessor (..) operator to access child properties of an XML object. Consider the following XML object:

```
var myXML:XML =
  <order>
    <book ISBN="0942407296">
      <title>Baking Extravagant Pastries with Kumquats</title>
      <author>
        <lastName>Contino</lastName>
        <firstName>Chuck</firstName>
      </author>
      <pageCount>238</pageCount>
    </book>
    <book ISBN="0865436401">
      <title>Emu Care and Breeding</title>
      <editor>
        <lastName>Case</lastName>
        <firstName>Justin</firstName>
      </editor>
      <pageCount>115</pageCount>
    </book>
  </order>
```

The object `myXML.book` is an XMLList object containing child properties of the `myXML` object that have the name `book`. These are two XML objects, matching the two `book` properties of the `myXML` object.

The object `myXML..lastName` is an XMLList object containing any descendent properties with the name `lastName`. These are two XML objects, matching the two `lastName` of the `myXML` object.

The object `myXML.book.editor.lastName` is an XMLList object containing any children with the name `lastName` of children with the name `editor` of children with the name `book` of the `myXML` object: in this case, an XMLList object containing only one XML object (the `lastName` property with the value "`Case`").

Accessing parent and child nodes

The `parent()` method returns the parent of an XML object.

You can use the ordinal index values of a child list to access specific child objects. For example, consider an XML object `myXML` that has two child properties named `book`. Each child property named `book` has an index number associated with it:

```
myXML.book[0]
myXML.book[1]
```

To access a specific grandchild, you can specify index numbers for both the child and grandchild names:

```
myXML.book[0].title[0]
```

However, if there is only one child of `x.book[0]` that has the name `title`, you can omit the index reference, as follows:

```
myXML.book[0].title
```

Similarly, if there is only one book child of the object `x`, and if that child object has only one title object, you can omit both index references, like this:

```
myXML.book.title
```

You can use the `child()` method to navigate to children with names based on a variable or expression, as the following example shows:

```
var myXML:XML = 
    <order>
      <book>
        <title>Dictionary</title>
      </book>
    </order>;

var childName:String = "book";

trace(myXML.child(childName).title) // output: Dictionary
```

Accessing attributes

Use the `@` symbol (the attribute identifier operator) to access attributes in an XML or XMLList object, as shown in the following code:

```
var employee:XML =
  <employee id="6401" code="233">
    <lastName>Wu</lastName>
    <firstName>Erin</firstName>
  </employee>;
trace(employee.@id); // 6401
```

You can use the `*` wildcard symbol with the `@` symbol to access all attributes of an XML or XMLList object, as in the following code:

```
var employee:XML =
  <employee id="6401" code="233">
    <lastName>Wu</lastName>
    <firstName>Erin</firstName>
  </employee>;
trace(employee.@*.toXMLString());
// 6401
// 233
```

You can use the `attribute()` or `attributes()` method to access a specific attribute or all attributes of an XML or XMLList object, as in the following code:

```
var employee:XML =
  <employee id="6401" code="233">
    <lastName>Wu</lastName>
    <firstName>Erin</firstName>
  </employee>;
trace(employee.attribute("id")); // 6401
trace(employee.attribute("*").toXMLString());
// 6401
// 233
trace(employee.attributes().toXMLString());
// 6401
// 233
```

Note that you can also use the following syntax to access attributes, as the following example shows:

```
employee.attribute("id")
employee["@id"]
employee.@["id"]
```

These are each equivalent to `employee.@id`. However, the syntax `employee.@id` is the preferred approach.

Filtering by attribute or element value

You can use the parentheses operators— `(` and `)` —to filter elements with a specific element name or attribute value. Consider the following XML object:

```
var x:XML =
  <employeeList>
    <employee id="347">
      <lastName>Zmed</lastName>
      <firstName>Sue</firstName>
      <position>Data analyst</position>
    </employee>
    <employee id="348">
      <lastName>McGee</lastName>
      <firstName>Chuck</firstName>
      <position>Jr. data analyst</position>
    </employee>
  </employeeList>
```

The following expressions are all valid:

- `x.employee.(lastName == "McGee")`—This is the second `employee` node.
- `x.employee.(lastName == "McGee").firstName`—This is the `firstName` property of the second `employee` node.
- `x.employee.(lastName == "McGee").@id`—This is the value of the `id` attribute of the second `employee` node.
- `x.employee.(@id == 347)`—The first `employee` node.
- `x.employee.(@id == 347).lastName`—This is the `lastName` property of the first `employee` node.
- `x.employee.(@id > 300)`—This is an XMLList with both `employee` properties.
- `x.employee.(position.toString().search("analyst") > -1)`—This is an XMLList with both `position` properties.

If you try to filter on attributes or elements that may not exist, Flash Player will throw an exception. For example, the final line of following code generates an errors because there is no `id` attribute in the second `p` element:

```
var doc:XML =
      <body>
        <p id='123'>Hello, <b>Bob</b>.</p>
        <p>Hello.</p>
      </body>;
trace(doc.p.(@id == '123'));
```

Similarly, the final line of following code generates an error because there is no `b` property of the second `p` element:

```
var doc:XML =
        <body>
           <p id='123'>Hello, <b>Bob</b>.</p>
           <p>Hello.</p>
        </body>;
trace(doc.p.(b == 'Bob'));
```

To avoid these errors, you can identify the properties that have the matching attributes or elements by using the `attribute()` and `elements()` methods, as in the following code:

```
var doc:XML =
        <body>
           <p id='123'>Hello, <b>Bob</b>.</p>
           <p>Hello.</p>
        </body>;
trace(doc.p.(attribute('id') == '123'));
trace(doc.p.(elements('b') == 'Bob'));
```

You can also use the `hasOwnProperty()` method, as in the following code:

```
var doc:XML =
        <body>
           <p id='123'>Hello, <b>Bob</b>.</p>
           <p>Hello.</p>
        </body>;
trace(doc.p.(hasOwnProperty('@id') && @id == '123'));
trace(doc.p.(hasOwnProperty('b') && b == 'Bob'));
```

Using the for..in and the for each..in statements

ActionScript 3.0 includes the `for..in` statement and the `for each..in` statement for iterating through XMLList objects. For example, consider the following XML object, `myXML`, and the XMLList object, `myXML.item`. The XMLList object, `myXML.item,` consists of the two `item` nodes of the XML object.

```
var myXML:XML =
   <order>
      <item id='1' quantity='2'>
         <menuName>burger</menuName>
         <price>3.95</price>
      </item>
      <item id='2' quantity='2'>
         <menuName>fries</menuName>
         <price>1.45</price>
      </item>
   </order>;
```

The `for..in` statement lets you iterate over a set of property names in an XMLList:

```
var total:Number = 0;
for (var pname:String in myXML.item)
{
  total += myXML.item.@quantity[pname] * myXML.item.price[pname];
}
```

The `for each..in` statement lets you iterate through the properties in the XMLList:

```
var total2:Number = 0;
for each (var prop:XML in myXML.item)
{
  total2 += prop.@quantity * prop.price;
}
```

Using XML namespaces

Namespaces in an XML object (or document) identify the type of data that the object contains. For example, in sending and delivering XML data to a web service that uses the SOAP messaging protocol, you declare the namespace in the opening tag of the XML:

```
var message:XML =
  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap:Body xmlns:w="http://www.test.com/weather/">
      <w:getWeatherResponse>
        <w:tempurature >78</w:tempurature>
      </w:getWeatherResponse>
    </soap:Body>
  </soap:Envelope>;
```

The namespace has a prefix, `soap`, and a URI that defines the namespace, `http://schemas.xmlsoap.org/soap/envelope/`.

ActionScript 3.0 includes the Namespace class for working with XML namespaces. For the XML object in the previous example, you can use the Namespace class as follows:

```
var soapNS:Namespace = message.namespace("soap");
trace(soapNS); // Output: http://schemas.xmlsoap.org/soap/envelope/

var wNS:Namespace = new Namespace("w", "http://www.test.com/weather/");
message.addNamespace(wNS);
var encodingStyle:XMLList = message.@soapNS::encodingStyle;
var body:XMLList = message.soapNS::Body;

message.soapNS::Body.wNS::GetWeatherResponse.wNS::tempurature = "78";
```

The XML class includes the following methods for working with namespaces: `addNamespace()`, `inScopeNamespaces()`, `localName()`, `name()`, `namespace()`, `namespaceDeclarations()`, `removeNamespace()`, `setLocalName()`, `setName()`, and `setNamespace()`.

The `default xml namespace` directive lets you assign a default namespace for XML objects. For example, in the following, both `x1` and `x2` have the same default namespace:

```
             var ns1:Namespace = new Namespace("http://www.example.com/
  namespaces/");
      default xml namespace = ns1;
var x1:XML = <test1 />;
      var x2:XML = <test2 />;
```

XML type conversion

You can convert XML objects and XMLList objects to String values. Similarly, you can convert strings to XML objects and XMLList objects. Also, keep in mind that all XML attribute values, names, and text values are strings. The following sections discuss all these forms of XML type conversion.

Converting XML and XMLList objects to strings

The XML and XMLList classes include a `toString()` method and a `toXMLString()` method. The `toXMLString()` method returns a string that includes all tags, attributes, namespace declarations, and content of the XML object. For XML objects with complex content (child elements), the `toString()` method does exactly the same as the `toXMLString()` method. For XML objects with simple content (those that contain only one text element), the `toString()` method returns only the text content of the element, as the following example shows:

```
var myXML:XML =
  <order>
    <item id='1' quantity='2'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
  <order>;

trace(myXML.item[0].menuName.toXMLString());
  // <menuName>burger</menuName>
trace(myXML.item[0].menuName.toString());
  // burger
```

If you use the `trace()` method without specifying `toString()` or `toXMLString()`, the data is converted using the `toString()` method by default, as this code shows:

```
var myXML:XML =
  <order>
    <item id='1' quantity='2'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
  <order>;

trace(myXML.item[0].menuName);
  // burger
```

When using the `trace()` method to debug code, you will often want to use the `toXMLString()` method so that the `trace()` method outputs more complete data.

Converting strings to XML objects

You can use the `new XML()` constructor to create an XML object from a string, as follows:

```
var x:XML = new XML("<a>test</a>");
```

If you attempt to convert a string to XML from a string that represents invalid XML or XML that is not well formed, a run-time error is thrown, as follows:

```
var x:XML = new XML("<a>test"); // throws an error
```

Converting attribute values, names, and text values from strings

All XML attribute values, names, and text values are String data types, and you may need to convert these to other data types. For example, the following code uses the `Number()` function to convert text values to numbers:

```
var myXML:XML =
             <order>
               <item>
                 <price>3.95</price>
               </item>
               <item>
                 <price>1.00</price>
               </item>
             </order>;

var total:XML = <total>0</total>;
myXML.appendChild(total);

for each (var item:XML in myXML.item)
```

```
{
  myXML.total.children()[0] = Number(myXML.total.children()[0])
                            + Number(item.price.children()[0]);
}
trace(myXML.total); // 4.35;
```

If this code did not use the `Number()` function, the code would interpret the + operator as the string concatenation operator, and the `trace()` method in the last line would output the following:

```
01.003.95
```

Reading external XML documents

You can use the URLLoader class to load XML data from a URL. To use the following code in your applications, replace the `XML_URL` value in the example with a valid URL:

```
var myXML:XML = new XML();
var XML_URL:String = "http://www.example.com/Sample3.xml";
var myXMLURL:URLRequest = new URLRequest(XML_URL);
var myLoader:URLLoader = new URLLoader(myXMLURL);
myLoader.addEventListener("complete", xmlLoaded);

function xmlLoaded(evtObj:Event):void
{
  myXML = XML(myLoader.data);
  trace("Data loaded.");
}
```

You can also use the XMLSocket class to set up an asynchronous XML socket connection with a server. For more information, see the *ActionScript 3.0 Language Reference*.

Example: Loading RSS data from the Internet

The RSSViewer sample application shows a number of features of working with XML in ActionScript, including the following:

- Using XML methods to traverse XML data in the form of an RSS feed.
- Using XML methods to assemble XML data in the form of HTML to use in a text field.

The RSS format is widely used to syndicate news via XML. A simple RSS data file may look like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
<channel>
  <title>Alaska - Weather</title>
  <link>http://www.nws.noaa.gov/alerts/ak.html</link>
  <description>Alaska - Watches, Warnings and Advisories</description>

  <item>
    <title>
      Short Term Forecast - Taiya Inlet, Klondike Highway (Alaska)
    </title>
    <link>
      http://www.nws.noaa.gov/alerts/ak.html#A18.AJKNK.1900
    </link>
    <description>
      Short Term Forecast Issued At: 2005-04-11T19:00:00
      Expired At: 2005-04-12T01:00:00 Issuing Weather Forecast Office
      Homepage: http://pajk.arh.noaa.gov
    </description>
  </item>
  <item>
    <title>
      Short Term Forecast - Haines Borough (Alaska)
    </title>
    <link>
      http://www.nws.noaa.gov/alerts/ak.html#AKZ019.AJKNOWAJK.190000
    </link>
    <description>
      Short Term Forecast Issued At: 2005-04-11T19:00:00
      Expired At: 2005-04-12T01:00:00 Issuing Weather Forecast Office
      Homepage: http://pajk.arh.noaa.gov
    </description>
  </item>
</channel>
</rss>
```

The SimpleRSS application reads RSS data from the Internet, parses the data for headlines (titles), links, and descriptions, and returns that data. The SimpleRSSUI class provides the UI and calls the SimpleRSS class, which does all of the XML processing.

The RSSViewer application files can be found in the folder Samples/RSSViewer. The application consists of the following files:

File	Description
RSSViewer.mxml	The main application file in MXML for Flex.
com/example/programmingas3/rssViewer/ RSSParser.as	A class that contains methods that use E4X to traverse RSS (XML) data and generate a corresponding HTML representation.
RSSData/ak.rss	A sample RSS file. The application is set up to read RSS data from the web, at a Flex RSS feed hosted by Adobe. However, you can easily change the application to read RSS data from this document, which uses a slightly different schema than that of the Flex RSS feed.

Reading and parsing XML data

The RSSParser class includes an `xmlLoaded()` method that converts the input RSS data, stored in the `rssXML` variable, into an string containing HTML-formatted output, `rssOutput`.

Near the beginning of the method, code sets the default XML namespace if the source RSS data includes a default namespace:

```
if (rssXML.namespace("") != undefined)
{
  default xml namespace = rssXML.namespace("");
}
```

The next lines then loop through the contents of the source XML data, examining each descendant property named `item`:

```
for each (var item:XML in rssXML..item)
{
  var itemTitle:String = item.title.toString();
  var itemDescription:String = item.description.toString();
  var itemLink:String = item.link.toString();
  outXML += buildItemHTML(itemTitle,
              itemDescription,
              itemLink);
}
```

The first three lines simply set string variables to represent the title, description and link properties of the `item` property of the XML data. The next line then calls the `buildItemHTML()` method to get HTML data in the form of an XMLList object, using the three new string variables as parameters.

Assembling XMLList data

The HTML data (an XMLList object) is of the following form:

```
<b>itemTitle</b>
<p>
  itemDescription
  <br />
  <a href="link">
    <font color="#008000">More...</font>
  </a>
</p>
```

The first lines of the method clear the default xml namespace:

```
default xml namespace = new Namespace();
```

The `default xml namespace` directive has function block-level scope. This means that the scope of this declaration is the `buildItemHTML()` method.

The lines that follow assemble the XMLList, based on the string arguments passed to the function:

```
var body:XMLList = new XMLList();
body += new XML("<b>" + itemTitle + "</b>");
var p:XML = new XML("<p>" + itemDescription + "</p>");

var link:XML = <a></a>;
link.@href = itemLink; // <link href="itemLinkString"></link>
link.font.@color = "#008000";
    // <font color="#008000"></font></a>
    // 0x008000 = green
link.font = "More...";

p.appendChild(<br/>);
p.appendChild(link);
body += p;
```

This XMLList object represents string data suitable for an ActionScript HTML text field.

The `xmlLoaded()` method uses the return value of the `buildItemHTML()` method and converts it to a string:

```
XML.prettyPrinting = false;
rssOutput = outXML.toXMLString();
```

Extracting the title of the RSS feed and sending a custom event

The `xmlLoaded()` method sets a `rssTitle` string variable, based on information in the source RSS XML data:

```
rssTitle = rssXML.channel.title.toString();
```

Finally, the `xmlLoaded()` method generates an event, which notifies the application that the data is parsed and available:

```
        dataWritten = new Event("dataWritten", true);
```

PART 3

Flash Player APIs

3

This part provides in-depth strategies for working with important features that are implemented in packages and classes specific to Adobe Flash Player 9.

The following chapters are included:

CHAPTER 12

Flash Player API Overview

12

This chapter describes the Adobe Flash Player 9 packages provided with the ActionScript 3.0 language.

The Flash Player API refers to all packages, classes, functions, properties, constants, events, and errors that are in the flash package. They are unique to Flash Player, as opposed to the top-level classes, such as Date, Math, and XML, or the language elements, which are based on ECMAScript. The Flash Player API contains features that you expect to find in object-oriented programming languages, such as the flash.geom package for geometry classes, as well as features specific to the needs of rich Internet applications, such as the flash.filters package for expressiveness and the flash.net package for handling data transmission to and from a server.

The Flash Player API in ActionScript 3.0 contains many new classes that allow you to control objects and data at a low level. The architecture of the language is completely new and more intuitive. For example, the API is organized into logical packages: the flash.display package contains all classes related to the visual display list in Flash Player; the flash.media package contains classes for working with audio and video; and flash.text contains classes for working with text in Flash Player applications.

This chapter provides a brief overview of each package and highlights the most important or commonly used classes and functions in the package. For detailed information about the classes or package-level functions, see the specific entry in the *ActionScript 3.0 Language Reference*.

Contents

flash.accessibility package

The flash.accessibility package contains classes for supporting accessibility in Flash content and applications. The Accessibility class manages communication with screen readers. The AccessibilityProperties class lets you control the presentation of Flash objects to screen readers and other accessibility aids. For more information, see the flash.accessibility package in the *ActionScript 3.0 Language Reference*.

flash.display package

The flash.display package contains the core classes that Flash Player uses to build visual displays and control objects on the display list, which contains all visual elements in a Flash Player application.

In ActionScript 2.0, nearly all visual elements were controlled by the MovieClip class or handled "behind the scenes" by Flash Player. Low-level control of objects was not available. In ActionScript 3.0, the API to control visual elements is defined more logically, by functionality and usage. The flash.display package includes the following classes:

- Basic building blocks that inherit from the DisplayObject class, such as DisplayObjectContainer and Sprite. MovieClip, also a basic building block, is used for objects that require a timeline. Functionality specific to buttons, such as properties for various button states, is contained in the SimpleButton class.
- Classes used to create expressive graphics, such as Graphics and Shape for vector graphics and Bitmap and BitmapData for bitmap images.
- Classes for timeline-based applications and animation, such as MovieClip and Scene.
- The Loader and LoaderInfo class, which handle loading of SWF files or image files (JPG, PNG, or GIF).
- Additional classes to support the classes just listed.

For more information, see "Display Programming" on page 159 and the flash.display package in the *ActionScript 3.0 Language Reference*.

flash.errors package

The flash.errors package contains error classes that relate to Flash Player-specific functionality, such as IOError (input/out error) and IllegalOperationError. In ActionScript 3.0, exceptions are the primary mechanism for reporting run-time errors. For more information, see "Handling Errors" on page 255 and the flash.errors package in the *ActionScript 3.0 Language Reference*.

flash.events package

The flash.events package supports the new XML Document Object Model (DOM) event model and includes the EventDispatcher base class. Events include error events. Error events are used when errors are encountered during an asynchronous operation, such as a call to the `Loader.load()` method. For more information, see "Handling Events" on page 345 and the flash.events package in the *ActionScript 3.0 Language Reference*.

flash.external package

The flash.external package contains only the ExternalInterface class, which was introduced in Flash Player 8 as a replacement for the `fscommand()` function. ExternalInterface enables communication between ActionScript and the Flash Player container—for example, an HTML page with JavaScript or a desktop application in which Flash Player is embedded. For more information, see "Using the External API" on page 501 and the flash.external package in the *ActionScript 3.0 Language Reference*.

flash.filters package

The flash.filters package contains classes for bitmap filter effects that were introduced in Flash Player 8. Filters let you apply rich visual effects, such as blur, bevel, glow, and drop shadows, to display objects. The classes in the flash.filters package are BevelFilter, BlurFilter, DisplacementMapFilter, GlowFilter, GradientBevelFilter, and GradientGlowFilter. This package also includes classes that offer a more complicated and varied range of effects by letting you apply matrixes to individual pixel values: ColorMatrixFilter and ConvolutionFilter. For more information, see the flash.filters package in the *ActionScript 3.0 Language Reference*.

flash.geom package

The flash.geom package contains geometry classes, such as Point, Rectangle, and Matrix, to support the Bitmap class, as well as the bitmap caching property of display objects. It also contains the Transform and ColorTransform classes for manipulating color values. For more information, see "Working with Geometry" on page 417 and the flash.geom package in the *ActionScript 3.0 Language Reference*.

flash.media package

The flash.media package contains classes for working with audio and video streams that are either prerecorded or that stream from the client computer that is running Flash Player. The Sound class and its supporting classes let you work with external MP3 files and streaming sound embedded in a SWF file. The Microphone class lets you capture audio from a microphone attached to the computer. The Camera and Video classes let you capture video from a video camera attached to the computer. The Video class also lets you work with prerecorded external FLV files. For more information, see the flash.media package in the *ActionScript 3.0 Language Reference*.

flash.net package

The flash.net package contains a variety of classes that handle the sending and receiving of data. Flash Player 9 can handle many kinds of data—raw binary data, XML, text, URL-encoded variables, and whole files that can be uploaded or downloaded. Data can be transmitted from a network server to the client computer that is running Flash Player, and vice-versa, or between two SWF files.

Central to the package are the URLLoader class and the URLRequest class. You use the URLLoader class to send variables in an object to a specified URL and to load variables at a specified URL into an object. The URLRequest class captures all the data in an HTTP request.

This package also contains package-level functions, such as `navigateToURL()` and `sendToURL()`, that you can use for simple operations that don't require much server interaction.

For more information, see "Networking and Communication" on page 371 and the flash.net package in the *ActionScript 3.0 Language Reference*.

flash.printing package

This package contains classes for printing Flash-based content. For more information, see "Printing" on page 487 and the flash.printing package in the *ActionScript 3.0 Language Reference*.

flash.profiler package

This package contains the `showRedrawRegions()` function, which can be useful for debugging code. For more information, see the flash.profiler package in the *ActionScript 3.0 Language Reference*.

flash.system package

The flash.system package contains classes that provide system-level functionality and let you get limited, specified information about the functions of the client computer running Flash Player. The Capabilities and IME classes let you determine certain hardware and software capabilities of the computer that is running Flash Player. Included in this package are classes pertaining to security: Security, SecurityDomain, and LoaderContext.

The flash.system package also contains the ApplicationDomain class. This class lets you partition your custom classes into separate application domains and to reuse definitions.

For more information, see "Client System Environment" on page 433 and the flash.system package in the *ActionScript 3.0 Language Reference*.

flash.text package

This package provides classes to work with text fields, text formatting, fonts, anti-aliasing, text metrics, style sheets, and layout. Advanced anti-aliasing is available in Flash Player 9 through the TextFormat and TextRenderer classes. Various new methods in the TextField class let you get low-level details about text fields. For more information, see the flash.text package in the *ActionScript 3.0 Language Reference*.

flash.ui package

The flash.ui package provides classes that let you customize the user interface, specifically the mouse cursor, the computer keyboard, and the context menu. For more information, see the flash.ui package in the *ActionScript 3.0 Language Reference*.

flash.utils package

This package provides various utility classes, such as ByteArray, which lets you access and work with data on the byte level, and the Timer class, which lets you run code on a specified time sequence. This package also contains a number of package-level functions that can control the time delay or intervals in which ActionScript code is executed. For more information, see the flash.utils package in the *ActionScript 3.0 Language Reference*.

flash.xml package

The flash.xml package provides legacy XML support. Because of the introduction of the top-level E4X-compliant XML class, the legacy XML-related classes that are compatible with ActionScript 1.0 and 2.0 XML objects are in the flash.xml package. For more information, see "Working with XML" on page 311 and the XML and XMLList packages in the *ActionScript 3.0 Language Reference*.

CHAPTER 13

Handling Events

13

An event handling system allows programmers to respond to user input and system events in a convenient way. The new ActionScript 3.0 event model is not only convenient, but also standards-compliant, and well integrated with the new Adobe Flash Player 9 display list. Based on the Document Object Model (DOM) Level 3 Events Specification, an industry-standard event handling architecture, the new event model provides a powerful yet intuitive event handling tool for ActionScript programmers.

This chapter is organized in five sections. The first two sections provide background information about event handling in ActionScript. The last three sections describe the main concepts behind the new event model: the event flow, the event object, and event listeners. The ActionScript 3.0 event handling system interacts closely with the display list, and this chapter assumes that you have a basic understanding of the display list. For more information, see "Display Programming" on page 159.

Contents

Introduction to Event Handling

You can think of events as occurrences of any kind in your SWF file that are of interest to you as a programmer. For example, most SWF files support user interaction of some sort—whether it's something as simple as responding to a mouse click or something more complex, such as accepting and processing data entered into a form. Any such user interaction with your SWF file is considered an event. Events can also occur without any direct user interaction, such as when data has finished loading from a server or when an attached camera has become active.

In ActionScript 3.0, each event is represented by an event object, which is an instance of the Event class or one of its subclasses. An event object not only stores information about a specific event, but also contains methods that facilitate manipulation of the event object. For example, when Flash Player detects a mouse click, it creates an event object to represent that particular mouse click event. In this case, the event object is an instance of the MouseEvent class.

After creating an event object, Flash Player *dispatches* it, which means that the event object is passed to the object that is the target of the event. An object that serves as the destination for a dispatched event object is called an *event target*. For example, when an attached camera becomes active, Flash Player dispatches an event object directly to the event target, which in this case is the object that represents the camera. If the event target is on the display list, however, the event object is passed down through the display list hierarchy until it reaches the event target. In some cases, the event object then "bubbles" back up the display list hierarchy along the same route. This traversal of the display list hierarchy is called the *event flow*.

You can "listen" for event objects in your code using event listeners. *Event listeners* are the functions or methods that you write to respond to specific events. To ensure that your program responds to events, you must add event listeners either to the event target or to any display list object that is part of an event object's event flow.

These three basic ideas form the structure of the new event model: the event flow, event objects, and event listeners. A solid understanding of each concept is important if you want to respond to events that occur in your SWF file. For advanced developers, the new event model allows you to create and dispatch custom events, but this chapter focuses primarily on the events that are defined by ActionScript 3.0 and dispatched directly by Flash Player.

How ActionScript 3.0 event handling differs from earlier versions

The most noticeable difference between event handling in ActionScript 3.0 and event handling in previous versions of ActionScript is that in ActionScript 3.0 there is only one system for event handling, whereas in previous versions of ActionScript there are several different event handling systems. This section begins with an overview of how event handling worked in previous versions of ActionScript, and then discusses how event handling has changed for ActionScript 3.0.

Event handling in previous versions of ActionScript

Versions of ActionScript before ActionScript 3.0 provided a number of different ways to handle events:

- `on()` event handlers that can be placed directly on Button and MovieClip instances
- `onClipEvent()` handlers that can be placed directly on MovieClip instances
- Callback function properties, such as `XML.onload` and `Camera.onActivity`
- Event listeners that you register using the `addListener()` method
- The UIEventDispatcher class that partially implemented the DOM event model.

Each of these mechanisms presents its own set of advantages and limitations. The `on()` and `onClipEvent()` handlers are easy to use, but make subsequent maintenance of projects more difficult because code placed directly on buttons and movie clips can be difficult to find. Callback functions are also simple to implement, but limit you to only one callback function for any given event. Event listeners are more difficult to implement—they require not only the creation of a listener object and function, but also the registration of the listener with the object that generates the event. This increased overhead, however, enables you to create several listener objects and register them all for the same event.

The development of components for ActionScript 2.0 engendered yet another event model. This new model, embodied in the UIEventDispatcher class, was based on a subset of the DOM Events Specification, and developers who are familiar with component event handling will find the transition to the new ActionScript 3.0 event model relatively painless.

Unfortunately, the syntax used by the various event models overlap in various ways, and differ in others. For example, in ActionScript 2.0, some properties, such as `TextField.onChanged`, can be used as either a callback function or an event listener. However, the syntax for registering listener objects differs depending on whether you are using one of the six classes that support listeners or the UIEventDispatcher class. For the Key, Mouse, MovieClipLoader, Selection, Stage, and TextField classes, you use the `addListener()` method, but for components event handling, you use a method called `addEventListener()`.

Another complexity introduced by the different event handling models was that the scope of the event handler function varied widely depending on the mechanism used. In other words, the meaning of the `this` keyword was not consistent among the event handling systems.

Event handling in ActionScript 3.0

ActionScript 3.0 introduces a single event handling model that replaces the many different event handling mechanisms that existed in previous versions of the language. The new event model is based on the Document Object Model (DOM) Level 3 Events Specification. Although the SWF file format does not adhere specifically to the Document Object Model standard, there are sufficient similarities between the display list and the structure of the DOM to make implementation of the DOM event model possible. An object on the display list is analogous to a node in the DOM hierarchical structure, and the terms *display list object* and *node* are used interchangeably throughout this discussion.

The Flash Player implementation of the DOM event model brings with it a new concept named default behaviors. A *default behavior* is an action that Flash Player executes as the normal consequence of certain events.

Default behaviors

Developers are usually responsible for writing code that responds to events. In some cases, however, a behavior is so commonly associated with an event that Flash Player automatically executes the behavior unless the developer adds code to cancel it. Because Flash Player automatically exhibits the behavior, such behaviors are called default behaviors.

For example, when a user enters text into a TextField object, the expectation that the text will be displayed in that TextField object is so common that the behavior is built into Flash Player. If you do not want this default behavior to occur, you can cancel it using the new event handling system. When a user inputs text into a TextField object, Flash Player creates an instance of the TextEvent class to represent that user input. To prevent Flash Player from displaying the text in the TextField object, you must access that specific TextEvent instance and call that instance's `preventDefault()` method.

Not all default behaviors can be prevented. For example, Flash Player generates a MouseEvent object when a user double-clicks a word in a TextField object. The default behavior, which cannot be prevented, is that the word under the cursor is highlighted.

Many types of event objects do not have associated default behaviors. For example, Flash Player dispatches a connect event object when a network connection is established, but there is no default behavior associated with it. The API documentation for the Event class and its subclasses lists each type of event and describes any associated default behavior, and whether that behavior can be prevented.

It is important to understand that default behaviors are associated only with event objects dispatched by Flash Player, and do not exist for event objects dispatched programmatically through ActionScript. For example, you can use the methods of the EventDispatcher class to dispatch an event object of type `textInput`, but that event object will not have a default behavior associated with it. In other words, Flash Player will not display a character in a TextField object as a result of a `textInput` event that you dispatched programmatically.

What's new for event listeners in ActionScript 3.0

For developers with experience using the ActionScript 2.0 `addListener()` method, it may be helpful to point out the differences between the ActionScript 2.0 event listener model and the ActionScript 3.0 event model. The following list describes a few major differences between the two event models:

- To add event listeners in ActionScript 2.0, you use `addListener()` in some cases and `addEventListener()` in others, whereas in ActionScript 3.0, you use `addEventListener()` in all situations.
- There is no event flow in ActionScript 2.0, which means that the `addListener()` method can be called only on the object that broadcasts the event, whereas in ActionScript 3.0, the `addEventListener()` method can be called on any object that is part of the event flow.
- In ActionScript 2.0, event listeners can be either functions, methods, or objects, whereas in ActionScript 3.0, only functions or methods can be event listeners.

The event flow

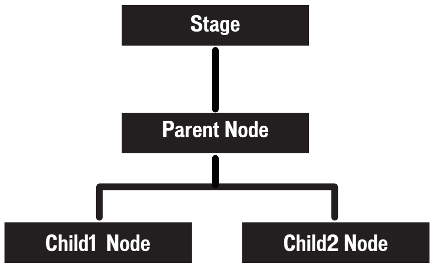
Flash Player dispatches event objects whenever an event occurs. If the event target is not on the display list, Flash Player dispatches the event object directly to the event target. For example, Flash Player dispatches the progress event object directly to a URLStream object. If the event target is on the display list, however, Flash Player dispatches the event object into the display list, and the event object travels through the display list to the event target.

The *event flow* describes how an event object moves through the display list. The display list is organized in a hierarchy that can be described as a tree. At the top of the display list hierarchy is the Stage, which is a special display object container that serves as the root of the display list. The Stage is represented by the flash.display.Stage class and can only be accessed through a display object. Every display object has a property named `stage` that refers to the Stage for that application.

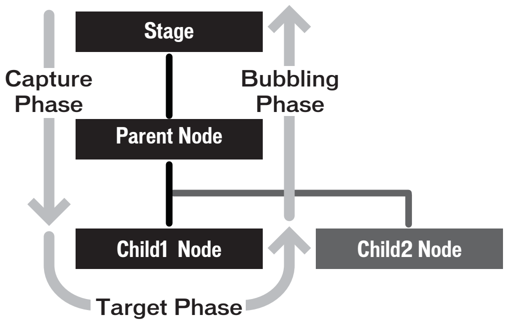
When Flash Player dispatches an event object, that event object makes a roundtrip journey from the Stage to the *target node*. The DOM Events Specification defines the target node as the node representing the event target. In other words, the target node is the display list object where the event occurred. For example, if a user clicks on a display list object named `child1`, Flash Player will dispatch an event object using `child1` as the target node.

The event flow is conceptually divided into three parts. The first part is called the capture phase; this phase comprises all of the nodes from the Stage to the parent of the target node. The second part is called the target phase, which consists solely of the target node. The third part is called the bubbling phase. The bubbling phase comprises the nodes encountered on the return trip from the parent of the target node back to the Stage.

The names of the phases make more sense if you conceive of the display list as a vertical hierarchy with the Stage at the top, as shown in the following diagram:



If a user clicks on `Child1 Node`, Flash Player dispatches an event object into the event flow. As the following image shows, the object's journey starts at `Stage`, moves down to `Parent Node`, then moves to `Child1 Node`, and then "bubbles" back up to `Stage`, moving through `Parent Node` again on its journey back to `Stage`.



In this example, the capture phase comprises `Stage` and `Parent Node` during the initial downward journey. The target phase comprises the time spent at `Child1 Node`. The bubbling phase comprises `Parent Node` and `Stage` as they are encountered during the upward journey back to the root node.

The event flow contributes to a more powerful event handling system than that previously available to ActionScript programmers. In previous versions of ActionScript, the event flow does not exist, which means that event listeners can be added only to the object that generates the event. In ActionScript 3.0, you can add event listeners not only to a target node, but also to any node along the event flow.

The ability to add event listeners along the event flow is useful when a user interface component comprises more than one object. For example, a button object often contains a text object that serves as the button's label. Without the ability to add a listener to the event flow, you would have to add a listener to both the button object and the text object to ensure that you receive notification about click events that occur anywhere on the button. The existence of the event flow, however, allows you to place a single event listener on the button object that handles click events that occur either on the text object or on the areas of the button object that are not obscured by the text object.

Not every event object, however, participates in all three phases of the event flow. Some types of events, such as the `enterFrame` and `init` event types, are dispatched directly to the target node and participate in neither the capture phase nor the bubbling phase. Other events may target objects that are not on the display list, such as events dispatched to an instance of the Socket class. These event objects will also flow directly to the target object, without participating in the capture and bubbling phases.

To find out how a particular event type behaves, you can either check the API documentation or examine the event object's properties. Examining the event object’s properties is described in the following section.

Event objects

Event objects serve two main purposes in the new event handling system. First, event objects represent actual events by storing information about specific events in a set of properties. Second, event objects contain a set of methods that allow you to manipulate event objects and affect the behavior of the event handling system.

To facilitate access to these properties and methods, the Flash Player API defines an Event class that serves as the base class for all event objects. The Event class defines a fundamental set of properties and methods that are common to all event objects.

This section begins with a discussion of the Event class properties, continues with a description of the Event class methods, and concludes with an explanation of why subclasses of the Event class exist.

Understanding Event class properties

The Event class defines a number of read-only properties and constants that provide important information about an event object.The following are especially important:

- Event object types are represented by constants and stored in the `Event.type` property.
- Whether an event’s default behavior can be prevented is represented by a Boolean value and stored in the `Event.cancelable` property.
- Event flow information is contained in the remaining properties.

Event object types

Every event object has an associated event type. Event types are stored in the `Event.type` property as string values. It is useful to know the type of an event object so that your code can distinguish objects of different types from one another. For example, the following code specifies that the `clickHandler()` listener function should respond to any mouse click event objects that are passed to `myDisplayObject`:

```
myDisplayObject.addEventListener(MouseEvent.CLICK, clickHandler);
```

Some two dozen event types are associated with the Event class itself and are represented by Event class constants, some of which are shown in the following excerpt from the Event class definition:

```
package flash.events
{
  public class Event
  {
    // class constants
    public static const ACTIVATE:String = "activate";
    public static const ADDED:String    = "added";
    // remaining constants omitted for brevity
  }
}
```

These constants provide an easy way to refer to specific event types. You should use these constants instead of the strings they represent. If you misspell a constant name in your code, the compiler will catch the mistake, but if you instead use strings, a typographical error may not manifest at compile time and could lead to unexpected behavior that could be difficult to debug. For example, when adding an event listener, use the following code:

```
myDisplayObject.addEventListener(MouseEvent.CLICK, clickHandler);
```

rather than:

```
myDisplayObject.addEventListener("click", clickHandler);
```

Default behavior information

Your code can check whether the default behavior for any given event object can be prevented by accessing the `cancelable` property. The `cancelable` property holds a Boolean value that indicates whether or not a default behavior can be prevented. You can prevent, or cancel, the default behavior associated with a small number of events using the `preventDefault()` method. For more information, see "Cancelling default event behavior" on page 356.

Event flow information

The remaining Event class properties contain important information about an event object and its relationship to the event flow, as described in the following list:

- The `bubbles` property contains information about the parts of the event flow in which the event object participates.
- The `eventPhase` property indicates the current phase in the event flow.
- The `target` property stores a reference to the event target.
- The `currentTarget` property stores a reference to the display list object that is currently processing the event object.

The bubbles property

An event is said to bubble if its event object participates in the bubbling phase of the event flow, which means that the event object is passed from the target node back through its ancestors until it reaches the Stage. The `Event.bubbles` property stores a Boolean value that indicates whether the event object participates in the bubbling phase. Because all events that bubble also participate in the capture and target phases, any event that bubbles participates in all three of the event flow phases. If the value is `true`, the event object participates in all three phases. If the value is `false`, the event object does not participate in the bubbling phase.

The eventPhase property

You can determine the event phase for any event object by investigating its `eventPhase` property. The `eventPhase` property contains an unsigned integer value that represents one of the three phases of the event flow. The Flash Player API defines a separate EventPhase class that contains three constants that correspond to the three unsigned integer values, as shown in the following code excerpt:

```
package flash.events
{
  public final class EventPhase
  {
    public static const CAPTURING_PHASE:uint = 1;
    public static const AT_TARGET:uint       = 2;
    public static const BUBBLING_PHASE:uint  = 3;
  }
}
```

These constants correspond to the three valid values of the `eventPhase` property. You can use these constants to make your code more readable. For example, if you want to ensure that a function named `myFunc()` is called only if the event target is in the target stage, you can use the following code to test for this condition:

```
if (e.eventPhase == EventPhase.AT_TARGET)
{
  myFunc();
}
```

The target property

The `target` property holds a reference to the object that is the target of the event. In some cases, this is straightforward, such as when a microphone becomes active, the target of the event object is the Microphone object. If the target is on the display list, however, the display list hierarchy must be taken into account. For example, if a user inputs a mouse click on a point that includes overlapping display list objects, Flash Player always chooses the object that is farthest away from the Stage as the event target.

For complex SWF files, especially those in which buttons are routinely decorated with smaller child objects, the `target` property may not be used frequently because it will often point to a button's child object instead of the button. In these situations, the common practice is to add event listeners to the button and use the `currentTarget` property because it points to the button, whereas the `target` property may point to a child of the button.

The currentTarget property

The `currentTarget` property contains a reference to the object that is currently processing the event object. Although it may seem odd not to know which node is currently processing the event object that you are examining, keep in mind that you can add a listener function to any display object in that event object's event flow, and the listener function can be placed in any location. Moreover, the same listener function can be added to different display objects. As a project increases in size and complexity, the `currentTarget` property becomes more and more useful.

Understanding Event class methods

There are three categories of Event class methods:

- Utility methods, which can create copies of an event object or convert it to a string
- Event flow methods, which remove event objects from the event flow
- Default behavior methods, which prevent default behavior or check whether it has been prevented

Event class utility methods

There are two utility methods in the Event class. The `clone()` method allows you to create copies of an event object. The `toString()` method allows you to generate a string representation of the properties of an event object along with their values. Both of these methods are used internally by the event model system, but are exposed to developers for general use.

For advanced developers creating subclasses of the Event class, you must override and implement versions of both utility methods to ensure that the event subclass will work properly.

Stopping event flow

You can call either the `Event.stopPropogation()` method or the `Event.stopImmediatePropogation()` method to prevent an event object from continuing on its way through the event flow. The two methods are nearly identical and differ only in whether the current node's other event listeners are allowed to execute:

- The `Event.stopPropogation()` method prevents the event object from moving on to the next node, but only after any other event listeners on the current node are allowed to execute.
- The `Event.stopImmediatePropogation()` method also prevents the event object from moving on to the next node, but does not allow any other event listeners on the current node to execute.

Calling either of these methods has no effect on whether the default behavior associated with an event occurs. Use the default behavior methods of the Event class to prevent default behavior.

Cancelling default event behavior

The two methods that pertain to cancelling default behavior are the `preventDefault()` method and the `isDefaultPrevented()` method. Call the `preventDefault()` method to cancel the default behavior associated with an event. To check whether `preventDefault()` has already been called on an event object, call the `isDefaultPrevented()` method, which returns a value of `true` if the method has already been called and `false` otherwise.

The `preventDefault()` method will work only if the event's default behavior can be cancelled. You can check whether this is the case by referring to the API documentation for that event type, or by using ActionScript to examine the `cancelable` property of the event object.

Cancelling the default behavior has no effect on the progress of an event object through the event flow. Use the event flow methods of the Event class to remove an event object from the event flow.

Subclasses of the Event class

For many events, the common set of properties defined in the Event class is sufficient. Other events, however, have unique characteristics that cannot be captured by the properties available in the Event class. For these events, the Flash Player API defines several subclasses of the Event class.

Each subclass provides additional properties and event types that are unique to that category of events. For example, events related to mouse input have several unique characteristics that cannot be captured by the properties defined in the Event class. The MouseEvent class extends the Event class by adding ten properties that contain information such as the location of the mouse event and whether specific keys were pressed during the mouse event.

An Event subclass also contains constants that represent the event types that are associated with the subclass. For example, the MouseEvent class defines constants for several mouse event types, include the `click`, `doubleClick`, `mouseDown`, and `mouseUp` event types.

As described in the section "Event class utility methods" on page 355, when creating an Event subclass you must override the `clone()` and `toString()` methods to provide functionality specific to the subclass.

Event listeners

Event listeners, which are also called event handlers, are functions that Flash Player executes in response to specific events. Adding an event listener is a two-step process. First, you create a function or class method for Flash Player to execute in response to the event. This is sometimes called the listener function or the event handler function. Second, you use the `addEventListener()` method to register your listener function with the target of the event or any display list object that lies along the appropriate event flow.

Creating a listener function

The creation of listener functions is one area where the ActionScript 3.0 event model deviates from the DOM event model. In the DOM event model, there is a clear distinction between an event listener and a listener function: an event listener is an instance of a class that implements the EventListener interface, whereas a listener function is a method of that class named `handleEvent()`. In the DOM event model, you register the class instance that contains the listener function rather than the actual listener function.

In the ActionScript 3.0 event model, there is no distinction between an event listener and a listener function. ActionScript 3.0 does not have an EventListener interface, and listener functions can be defined outside a class or as part of a class. Moreover, listener functions do not have to be named `handleEvent()`—they can be named with any valid identifier. In ActionScript 3.0, you register the name of the actual listener function.

Listener function defined outside of a class

The following code creates a simple SWF file that displays a red square shape. A listener function named `clickHandler()`, which is not part of a class, listens for mouse click events on the red square.

```
package
{
  import flash.display.Sprite;

  public class ClickExample extends Sprite
  {
    public function ClickExample()
    {
      var child:ChildSprite = new ChildSprite();
      addChild(child);
    }
  }
}

import flash.display.Sprite;
import flash.events.MouseEvent;

class ChildSprite extends Sprite
{
  public function ChildSprite()
  {
    graphics.beginFill(0xFF0000);
    graphics.drawRect(0,0,100,100);
    graphics.endFill();
    addEventListener(MouseEvent.CLICK, clickHandler);
  }
}

function clickHandler(event:MouseEvent):void
{
  trace("clickHandler detected an event of type: " + event.type);
  trace("the this keyword refers to: " + this);
}
```

When a user interacts with the resulting SWF file by clicking on the square, Flash Player generates the following trace output:

```
clickHandler detected an event of type: click
the this keyword refers to: [object global]
```

Notice that the event object is passed as an argument to `clickHandler()`. This allows your listener function to examine the event object. In this example, you use the event object's `type` property to ascertain that the event is a click event.

The example also checks the value of the `this` keyword. In this case, `this` represents the global object, which makes sense because the function is defined outside of any custom class or object.

Listener function defined as a class method

The following example is identical to the previous example that defines the ClickExample class except that the `clickHandler()` function is defined as a method of the ChildSprite class:

```
package
{
  import flash.display.Sprite;

  public class ClickExample extends Sprite
  {
    public function ClickExample()
    {
      var child:ChildSprite = new ChildSprite();
      addChild(child);
    }
  }
}

import flash.display.Sprite;
import flash.events.MouseEvent;

class ChildSprite extends Sprite
{
  public function ChildSprite()
  {
    graphics.beginFill(0xFF0000);
    graphics.drawRect(0,0,100,100);
    graphics.endFill();
    addEventListener(MouseEvent.CLICK, clickHandler);
  }
  private function clickHandler(event:MouseEvent):void
  {
    trace("clickHandler detected an event of type: " + event.type);
```

```
        trace("the this keyword refers to: " + this);
    }
}
```

When a user interacts with the resulting SWF file by clicking on the red square, Flash Player generates the following trace output:

```
clickHandler detected an event of type: click
the this keyword refers to: [object ChildSprite]
```

Note that the `this` keyword refers to the ChildSprite instance named `child`. This is a change in behavior from ActionScript 2.0. If you used components in ActionScript 2.0, you may remember that when a class method was passed in to `UIEventDispatcher.addEventListener()`, the scope of the method was bound to the component that broadcast the event instead of the class in which the listener method was defined. In other words, if you used this technique in ActionScript 2.0, the `this` keyword would refer to the component broadcasting the event instead of the ChildSprite instance.

This was a significant issue for some programmers because it meant that they could not access other methods and properties of the class containing the listener method. As a workaround, ActionScript 2.0 programmers could use the `mx.util.Delegate` class to change the scope of the listener method. This is no longer necessary, however, because ActionScript 3.0 creates a bound method when `addEventListener()` is called. As a result, the `this` keyword refers to the ChildSprite instance named `child`, and the programmer has access to the other methods and properties of the ChildSprite class.

Event listener that should not be used

There is a third technique in which you create a generic object with a property that points to a dynamically assigned listener function, but it is not recommended. It is discussed here because it was commonly used in ActionScript 2.0, but should not be used in ActionScript 3.0. This technique is not recommended because the `this` keyword will refer to the global object instead of your listener object.

The following example is identical to the previous ClickExample class example, except that the listener function is defined as part of a generic object named `myListenerObj`:

```
package
{
    import flash.display.Sprite;

    public class ClickExample extends Sprite
    {
        public function ClickExample()
        {
            var child:ChildSprite = new ChildSprite();
            addChild(child);
    }
  }
}

import flash.display.Sprite;
import flash.events.MouseEvent;

class ChildSprite extends Sprite
{
  public function ChildSprite()
  {
    graphics.beginFill(0xFF0000);
    graphics.drawRect(0,0,100,100);
    graphics.endFill();
    addEventListener(MouseEvent.CLICK, myListenerObj.clickHandler);
  }
}

var myListenerObj:Object = new Object();
myListenerObj.clickHandler = function (event:MouseEvent):void
{
    trace("clickHandler detected an event of type: " + event.type);
    trace("the this keyword refers to: " + this);
}
```

The results of the trace will look like this:

```
clickHandler detected an event of type: click
the this keyword refers to: [object global]
```

You would expect that `this` would refer to `myListenerObj` and that the trace output would be `[object Object]`, but instead it refers to the global object. When you pass in a dynamic property name as an argument to `addEventListener()`, Flash Player is unable to create a bound method. This is because what you are passing as the `listener` parameter is nothing more than the memory address of your listener function, and Flash Player has no way to link that memory address with the `myListenerObj` instance.

Managing event listeners

You can manage your listener functions using the methods of the IEventDispatcher interface. The IEventDispatcher interface is the ActionScript 3.0 version of the EventTarget interface of the DOM event model. Although the name IEventDispatcher may seem to imply that its main purpose is to send (or dispatch) event objects, the methods of this class are actually used much more frequently to register event listeners, check for event listeners, and remove event listeners. The IEventDispatcher interface defines five methods, as shown in the following code:

```
package flash.events
```

```
{
  public interface IEventDispatcher
  {
    function addEventListener(eventName:String,
            listener:Object,
            useCapture:Boolean=false,
            priority:Integer=0,
            useWeakReference:Boolean=false):Boolean;

    function removeEventListener(eventName:String,
          listener:Object,
          useCapture:Boolean=false):Boolean;

    function dispatchEvent(eventObject:Event):Boolean;

    function hasEventListener(eventName:String):Boolean;
    function willTrigger(eventName:String):Boolean;
  }
}
```

The Flash Player API implements the IEventDispatcher interface with the EventDispatcher class, which serves as a base class for all classes that can be event targets or part of an event flow. For example, the DisplayObject class inherits from the EventDispatcher class. This means that any object on the display list has access to the methods of the IEventDispatcher interface.

Adding event listeners

The `addEventListener()` method is the workhorse of the IEventDispatcher interface. You use it to register your listener functions. The two required parameters are `type` and `listener`. You use the `type` parameter to specify the type of event. You use the `listener` parameter to specify the listener function that will execute when the event occurs. The `listener` parameter can be a reference to either a function or a class method.

NOTE

Do not use parentheses when you specify the `listener` parameter. For example, the `clickHandler()` function is specified without parentheses in the following call to the `addEventListener()` method:
`addEventListener(MouseEvent.CLICK, clickHandler).`

The `useCapture` parameter of the `addEventListener()` method allows you to control the event flow phase on which your listener will be active. If `useCapture` is set to `true`, your listener will be active during the capture phase of the event flow. If `useCapture` is set to `false`, your listener will be active during the target and bubbling phases of the event flow. To listen for an event during all phases of the event flow, you must call `addEventListener()` twice, once with `useCapture` set to `true`, and then again with `useCapture` set to `false`.

The `priority` parameter of the `addEventListener()` method is not an official part of the DOM Level 3 event model. It is included in ActionScript 3.0 to provide you with more flexibility in organizing your event listeners. When you call `addEventListener()`, you can set the priority for that event listener by passing an integer value as the `priority` parameter. The default value is 0, but you can set it to negative or positive integer values. The higher the number, the sooner that event listener will be executed. Event listeners with the same priority are executed in the order that they were added, so the earlier a listener is added, the sooner it will be executed.

The `useWeakReference` parameter allows you to specify whether the reference to the listener function is weak or normal. Setting this parameter to `true` allows you to avoid situations in which listener functions persist in memory even though they are no longer needed. Flash Player uses a technique called *garbage collection* to clear objects from memory that are no longer in use. An object is considered no longer in use if no references to it exist. The garbage collector disregards weak references, which means that a listener function that has only a weak reference pointing to it is eligible for garbage collection.

Removing event listeners

You can use the `removeEventListener()` method to remove an event listener that you no longer need. It is a good idea to remove any listeners that will no longer be used. Required parameters include the `eventName` and `listener` parameters, which are the same as the required parameters for the `addEventListener()` method. Recall that you can listen for events during all event phases by calling `addEventListener()` twice, once with `useCapture` set to `true`, and then again with it set to `false`. To remove both event listeners, you would need to call `removeEventListener()` twice, once with `useCapture` set to `true`, and then again with it set to `false`.

Dispatching events

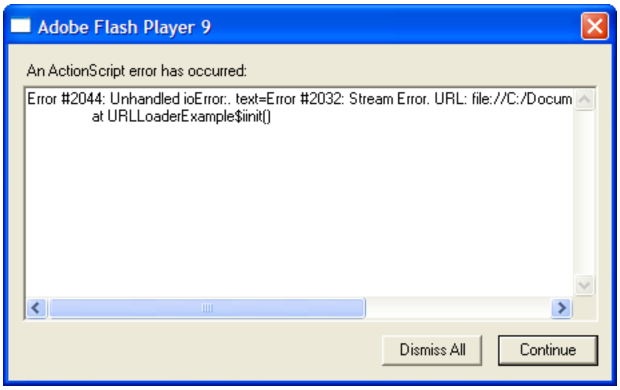
The `dispatchEvent()` method can be used by advanced programmers to dispatch a custom event object into the event flow. The only parameter accepted by this method is a reference to an event object, which must be an instance of the Event class or a subclass of the Event class. Once dispatched, the `target` property of the event object is set to the object on which `dispatchEvent()` was called.

Checking for existing event listeners

The final two methods of the IEventDispatcher interface provide useful information about the existence of event listeners. The `hasEventListener()` method returns `true` if an event listener is found for a specific event type on a particular display list object. The `willTrigger()` method also returns `true` if a listener is found for a particular display list object, but `willTrigger()` checks for listeners not only on that display object, but also on all of that display list object's ancestors for all phases of the event flow.

Error events without listeners

Exceptions, rather than events, are the primary mechanism for error handling in ActionScript 3.0, but exception handling does not work for asynchronous operations such as loading files. If an error occurs during such an asynchronous operation, Flash Player dispatches an error event object. If you do not create a listener for the error event, the debugger version of Flash Player will bring up a dialog box with information about the error. For example, using an invalid URL when loading a file produces this dialog box in the debugger version of Flash Player:



An ActionScript error has occurred:

```
Error #2044: Unhandled ioError:. text=Error #2032: Stream Error. URL: file://C:/Docum
	at URLLoaderExample$iinit()
```

Most error events are based on the ErrorEvent class, and as such will have a property named `text` that is used to store the error message that Flash Player displays. The two exceptions are the StatusEvent and NetStatusEvent classes. Both of these classes have a `level` property (`StatusEvent.level` and `NetStatusEvent.info.level`). When the value of the `level` property is `"error"`, these event types are considered to be error events.

An error event will not cause a SWF file to stop running. It will manifest only as a dialog box on the debugger versions of the browser plug-ins and stand-alone players, as a message in the output panel in the authoring player, and as an entry in the log file for Adobe Flex Builder 2. It will not manifest at all in the release versions of Flash Player.

Example: Alarm Clock

The Alarm Clock example consists of a clock that allows the user to specify a time at which an alarm will go off, as well as a message to be displayed at that time. The Alarm Clock example builds on the SimpleClock application from Chapter 6, "Working with Dates and Times." Alarm Clock illustrates several aspects of working with events in ActionScript 3.0, including:

- Listening and responding to an event
- Notifying listeners of an event
- Creating a custom event type

The Alarm Clock application files can be found in the Samples/AlarmClock folder. The application includes these files:

File	Description
AlarmClockApp.mxml	The main application file in MXML for Flex.
com/example/programmingas3/clock/AlarmClock.as	A class which extends the SimpleClock class, adding alarm clock functionality.
com/example/programmingas3/clock/AlarmEvent.as	A custom event class (a subclass of flash.events.Event) which serves as the event object for the AlarmClock class's `alarm` event.
com/example/programmingas3/clock/AnalogClockFace.as	Draws a round clock face and hour, minute, and seconds hands based on the time (described in the SimpleClock example).
com/example/programmingas3/clock/SimpleClock.as	A clock interface component with simple timekeeping functionality (described in the SimpleClock example).

Alarm Clock overview

The primary functionality of the clock in this example, including tracking the time and displaying the clock face, reuses the SimpleClock application code, which is described in "Example: Simple analog clock" on page 204. The AlarmClock class extends the SimpleClock class from that example by adding the functionality required for an alarm clock, including setting the alarm time and providing notification when the alarm "goes off."

Providing notification when something happens is the job that events are made for. The AlarmClock class exposes the Alarm event, which other objects can listen for in order to perform desired actions. In addition, the AlarmClock class uses an instance of the Timer class to determine when to trigger its alarm. Like the AlarmClock class, the Timer class provides an event to notify other objects (an AlarmClock instance, in this case) when a certain amount of time has passed. As with most ActionScript applications, events form an important part of the functionality of the Alarm Clock example application.

Triggering the alarm

As mentioned previously, the only functionality that the AlarmClock class actually provides relates to setting and triggering the alarm. The built-in Timer class (flash.utils.Timer) provides a way for a developer to define code that will be executed after a specified amount of time. The AlarmClock class uses a Timer instance to determine when to set off the alarm.

```
import flash.events.TimerEvent;
import flash.utils.Timer;

/**
 * The Timer that will be used for the alarm.
 */
public var alarmTimer:Timer;
...
/**
 * Instantiates a new AlarmClock of a given size.
 */
public override function initClock(faceSize:Number = 200):void
{
  super.initClock(faceSize);
  alarmTimer = new Timer(0, 1);
  alarmTimer.addEventListener(TimerEvent.TIMER, onAlarm);
}
```

The Timer instance defined in the AlarmClock class is named `alarmTimer`. The `initClock()` method, which performs necessary setup operations for the AlarmClock instance, does two things with the `alarmTimer` variable. First, the variable is instantiated with parameters instructing the Timer instance to wait 0 milliseconds and only trigger its timer event one time. After instantiating `alarmTimer`, the code calls that variable's `addEventListener()` method to indicate that it wants to listen to that variable's `timer` event. A Timer instance works by dispatching its `timer` event after a specified amount of time has passed. The AlarmClock class will need to know when the `timer` event is dispatched in order to set off its own alarm. By calling `addEventListener()`, the AlarmClock code registers itself as a listener with `alarmTimer`. The two parameters indicate that the AlarmClock class wants to listen for the `timer` event (indicated by the constant `TimerEvent.TIMER`), and that when the event happens, the AlarmClock class's `onAlarm()` method should be called in response to the event.

In order to actually set the alarm, the AlarmClock class's `setAlarm()` method is called, as follows:

```
/**
 * Sets the time at which the alarm should go off.
 * @param hour The hour portion of the alarm time.
 * @param minutes The minutes portion of the alarm time.
 * @param message The message to display when the alarm goes off.
 * @return The time at which the alarm will go off.
 */
public function setAlarm(hour:Number = 0, minutes:Number = 0,
message:String = "Alarm!"):Date
{
  this.alarmMessage = message;
  var now:Date = new Date();
  // Create this time on today's date.
  alarmTime = new Date(now.fullYear, now.month, now.date, hour, minutes);

  // Determine if the specified time has already passed today.
  if (alarmTime <= now)
  {
    alarmTime.setTime(alarmTime.time + MILLISECONDS_PER_DAY);
  }

  // Stop the alarm timer if it's currently set.
  alarmTimer.reset();
  // Calculate how many milliseconds should pass before the alarm should
  // go off (the difference between the alarm time and now) and set that
  // value as the delay for the alarm timer.
  alarmTimer.delay = Math.max(1000, alarmTime.time - now.time);
  alarmTimer.start();

  return alarmTime;
}
```

This method does several things, including storing the alarm message and creating a Date object (`alarmTime`) representing the actual moment in time when the alarm is to go off. Of most relevance to the current discussion, in the final several lines of the method, the `alarmTimer` variable's timer is set and activated. First, its `reset()` method is called, stopping the timer and resetting it in case it is already running. Next, the current time (represented by the `now` variable) is subtracted from the `alarmTime` variable's value to determine how many milliseconds need to pass before the alarm goes off. The Timer class doesn't trigger its `timer` event at an absolute time, so it is this relative time difference that is assigned to the `delay` property of `alarmTimer`. Finally, the `start()` method is called to actually start the timer.

Once the specified amount of time has passed, `alarmTimer` dispatches the `timer` event. Because the AlarmClock class registered its `onAlarm()` method as a listener for that event, when the `timer` event happens, `onAlarm()` is called.

```
/**
 * Called when the timer event is dispatched.
 */
public function onAlarm(event:TimerEvent):void
{
  trace("Alarm!");
  var alarm:AlarmEvent = new AlarmEvent(this.alarmMessage);
  this.dispatchEvent(alarm);
}
```

A method that is registered as an event listener must be defined with the appropriate signature (that is, the set of parameters and return type of the method). To be a listener for the Timer class's `timer` event, a method must define one parameter whose data type is TimerEvent (flash.events.TimerEvent), a subclass of the Event class. When the Timer instance calls its event listeners, it passes a TimerEvent instance as the event object.

Notifying others of the alarm

Like the Timer class, the AlarmClock class provides an event that allows other code to receive notifications when the alarm goes off. For a class to use the event handling framework built into ActionScript, that class must implement the flash.events.IEventDispatcher interface. Most commonly, this is done by extending the flash.events.EventDispatcher class, which provides a standard implementation of IEventDispatcher (or by extending one of EventDispatcher's subclasses). As described previously, the AlarmClock class extends the SimpleClock class, which in turn extends the mx.core.UIComponent class, which (through a chain of inheritance) extends the EventDispatcher class. All of this means that the AlarmClock class already has built-in functionality to provide its own events.

Other code can register to be notified of the AlarmClock class's `alarm` event by calling the `addEventListener()` method that AlarmClock inherits from EventDispatcher. When an AlarmClock instance is ready to notify other code that its `alarm` event has been raised, it does so by calling the `dispatchEvent()` method, which is also inherited from EventDispatcher.

```
var alarm:AlarmEvent = new AlarmEvent(this.alarmMessage);
this.dispatchEvent(alarm);
```

These lines of code are taken from the AlarmClock class's `onAlarm()` method (shown in its entirety previously). The AlarmClock instance's `dispatchEvent()` method is called, which in turn notifies all the registered listeners that the AlarmClock instance's `alarm` event has been triggered. The parameter that is passed to `dispatchEvent()` is the event object that will be passed along to the listener methods. In this case, it is an instance of the AlarmEvent class, an Event subclass created specifically for this example.

Providing a custom alarm event

All event listeners receive an event object parameter with information about the particular event being triggered. In many cases, the event object is an instance of the Event class. However, in some cases it is useful to provide additional information to event listeners. As described earlier in the chapter, a common way to accomplish this is to define a new class, a subclass of the Event class, and use an instance of that class as the event object. In this example, an AlarmEvent instance is used as the event object when the AlarmClock class's `alarm` event is dispatched. The AlarmEvent class, shown here, provides additional information about the `alarm` event, specifically the alarm message:

```
import flash.events.Event;

/**
 * This custom Event class adds a message property to a basic Event.
 */
public class AlarmEvent extends Event
{
    /**
     * The name of the new AlarmEvent type.
     */
    public static const ALARM:String = "alarm";

    /**
     * A text message that can be passed to an event handler
     * with this event object.
     */
    public var message:String;

    /**
     *  Constructor.
     *  @param message The text to display when the alarm goes off.
     */
    public function AlarmEvent(message:String = "ALARM!")
    {
      super(ALARM);
      this.message = message;
    }
    ...
}
```

The best way to create a custom event object class is to define a class that extends the Event class, as shown in the preceding example. To supplement the inherited functionality, the AlarmEvent class defines a property `message` that contains the text of the alarm message associated with the event; the `message` value is passed in as a parameter in the AlarmEvent constructor. The AlarmEvent class also defines the constant `ALARM`, which can be used to refer to the specific event (`alarm`) when calling the AlarmClock class's `addEventListener()` method.

In addition to adding custom functionality, every Event subclass must override the inherited `clone()` method as part of the ActionScript event handling framework. Event subclasses can also optionally override the inherited `toString()` method to include the custom event's properties in the value returned when the `toString()` method is called.

```
/**
 * Creates and returns a copy of the current instance.
 * @return A copy of the current instance.
 */
public override function clone():Event
{
  return new AlarmEvent(message);
}

/**
 * Returns a String containing all the properties of the current
 * instance.
 * @return A string representation of the current instance.
 */
public override function toString():String
{
  return formatToString("AlarmEvent", "type", "bubbles", "cancelable",
"eventPhase", "message");
}
```

The overridden `clone()` method needs to return a new instance of the custom Event subclass, with all the custom properties set to match the current instance. In the overridden `toString()` method, the utility method `formatToString()` (inherited from Event) is used to provide a string with the name of the custom type, as well as the names and values of all its properties.

CHAPTER 14

Networking and Communication

14

The flash.net package contains classes to send and receive data across the Internet—for example, to load content from remote URLs, to communicate with other Flash Player instances, and to connect to remote websites. In earlier versions of ActionScript, many of the classes within the flash.net package were top-level classes. In ActionScript 3.0, many of these classes have been moved into the flash.net package.

The flash.net package contains the following classes: FileReference, LocalConnection, NetConnection, NetStream, SharedObject, Socket, URLLoader, URLRequest, URLStream, URLVariables, and XMLSocket. flash.net contains four methods: `navigateToURL()`, `sendToURL()`, `registerClassAlias()`, and `getClassByAlias()`.

This chapter explains how to enable your SWF file to communicate with external files and other Adobe Flash Player 9 instances. It also explains how to load data from external sources, send messages between a Java server and Flash Player, and perform file uploads and downloads using the FileReference and FileReferenceList classes.

Contents

Working with external data

When you build large ActionScript applications, you often need to communicate with server-side scripts, or load data from external XML or text files. This behavior has changed significantly between ActionScript 2.0 and ActionScript 3.0. In earlier versions of ActionScript, you could load remote text files using the LoadVars class and the `LoadVars.onData()` event handler. In ActionScript 3.0, you can load external files with the URLLoader and URLRequest classes. If the remote content is formatted as name-value pairs, you use the URLVariables class to parse the server results.

You use the URLLoader and URLRequest classes to load the remote XML document. You can then parse the XML document using the XML class's constructor, the XMLDocument class's constructor, or the `XMLDocument.parseXML()` method. This allows you to simplify your ActionScript code because the code for loading external files is the same whether you use URLVariables or the XML classes.

Using the URLLoader and URLVariables classes

ActionScript 3.0 has replaced the LoadVars class with URLLoader and URLVariables classes. The URLLoader class downloads data from a URL as text, binary data, or URL-encoded variables. The URLLoader class is useful for downloading text files, XML, or other information to use in dynamic, data-driven ActionScript applications. The URLLoader class takes advantage of the ActionScript 3.0 advanced event handling model, which allows you to listen for such events as `complete`, `httpStatus`, `ioError`, `open`, `progress`, and `securityError`. The new event handling model is a significant improvement over the ActionScript 2.0 support for the `LoadVars.onData`, `LoadVars.onHTTPStatus`, and `LoadVars.onLoad` event handlers because it allows you to handle errors and events more efficiently. For more information on handling events, see Chapter 13, "Handling Events."

Much like the XML and LoadVars classes in earlier versions of ActionScript, the data of the URLLoader URL is not available until the download has completed. You can monitor the progress of the download (bytes loaded and bytes total) by listening for the `flash.events.ProgressEvent.PROGRESS` event to be dispatched, although if a file loads too quickly a `ProgressEvent.PROGRESS` event may not be dispatched. When a file has successfully downloaded, the `flash.events.Event.COMPLETE` event will be dispatched. The loaded data is decoded from UTF-8 or UTF-16 encoding into a string.

NOTE

If no value is set for `URLRequest.contentType`, values are sent as `application/x-www-form-urlencoded`.

The `URLLoader.load()` method (and optionally the URLLoader class's constructor) takes a single parameter, `request`, which is a URLRequest class object. A URLRequest object contains all of the information for a single HTTP request, such as the target URL, request method (`GET` or `POST`), additional header information, and the MIME type (for example, when you upload XML content).

For example, to upload an XML packet to a server-side script, you could use the following ActionScript 3.0 code:

```
var secondsUTC:Number = new Date().time;
var dataXML:XML = 
  <login>
    <time>{secondsUTC}</time>
    <username>Ernie</username>
    <password>guru</password>
  </login>;
var request:URLRequest = new URLRequest("http://www.yourdomain.com/
  login.cfm");
request.contentType = "text/xml";
request.data = dataXML.toXMLString();
request.method = URLRequestMethod.POST;
var loader:URLLoader = new URLLoader();
try
{
  loader.load(request);
}
catch (error:ArgumentError)
{
  trace("An ArgumentError has occurred.");
}
catch (error:SecurityError)
{
  trace("A SecurityError has occurred.");
}
```

The previous snippet creates an XML instance named `dataXml` that contains an XML packet to be sent to the server. Next, you set the URLRequest `contentType` property to `"text/xml"` and set the URLRequest `data` property to the contents of the XML packet, which are converted to a string by using the `XML.toXMLString()` method. Finally, you create a new URLLoader instance and send the request to the remote script by using the `URLLoader.load()` method.

There are three ways in which you can specify parameters to pass in a URL request:

- Within the URLVariables constructor.
- Within the `URLVariables.decode()` method.
- As specific properties within the URLVariables object itself.

When you define variables within the URLVariables constructor or within the `URLVariables.decode()` method, you need to make sure that you URL-encode the ampersand (`&`) character because it has a special meaning and acts as a delimiter. For example, when you pass an ampersand, you need to URL-encode the ampersand by changing it from `&` to `%26` because the ampersand acts as a delimiter for parameters.

Loading data from external documents

When you build dynamic applications with ActionScript 3.0, it's a good idea to load data from external files or from server-side scripts. This lets you build dynamic applications without having to edit or recompile your ActionScript files. For example, if you build a "tip of the day" application, you can write a server-side script that retrieves a random tip from a database and saves it to a text file once a day. Then your ActionScript application can load the contents of a static text file instead of querying the database each time.

The following snippet creates a URLRequest and URLLoader object, which loads the contents of an external text file, params.txt:

```
var request:URLRequest = new URLRequest("params.txt");
var loader:URLLoader = new URLLoader();
loader.load(request);
```

You can simplify the previous snippet to the following:

```
var loader:URLLoader = new URLLoader(new URLRequest("params.txt"));
```

By default, if you do not define a request method, Flash Player loads the content using the HTTP `GET` method. If you want to send the data using the `POST` method, you need to set the `request.method` property to `POST` using the static constant `URLRequestMethod.POST`, as the following code shows:

```
var request:URLRequest = new URLRequest("sendfeedback.cfm");
request.method = URLRequestMethod.POST;
```

The external document, params.txt, that is loaded at run time contains the following data:

```
monthNames=January,February,March,April,May,June,July,August,September,Octo
  ber,November,December&dayNames=Sunday,Monday,Tuesday,Wednesday,Thursday,
  Friday,Saturday
```

The file contains two parameters, `monthNames` and `dayNames`. Each parameter contains a comma-separated list that is parsed as strings. You can split this list into an array using the `String.split()` method.

TIP

Avoid using reserved words or language constructs as variable names in external data files, because doing so makes reading and debugging your code more difficult.

Once the data has loaded, the `Event.COMPLETE` event is dispatched, and the contents of the external document are available to use in the URLLoader's `data` property, as the following code shows:

```
private function completeHandler(event:Event):void
{
  var loader2:URLLoader = URLLoader(event.target);
  trace(loader2.data);
}
```

If the remote document contains name-value pairs, you can parse the data using the URLVariables class by passing in the contents of the loaded file, as follows:

```
private function completeHandler(event:Event):void
{
  var loader2:URLLoader = URLLoader(event.target);
  var variables:URLVariables = new URLVariables(loader2.data);
  trace(variables.dayNames);
}
```

Each name-value pair from the external file is created as a property in the URLVariables object. Each property within the variables object in the previous code sample is treated as a string. If the value of the name-value pair is a list of items, you can convert the string into an array by calling the `String.split()` method, as follows:

```
var dayNameArray:Array = variables.dayNames.split(",");
```

TIP: If you are loading numeric data from external text files, you need to convert the values into numeric values by using a top-level function, such as `int()`, `uint()`, or `Number()`.

Instead of loading the contents of the remote file as a string and creating a new URLVariables object, you could instead set the `URLLoader.dataFormat` property to one of the static properties found in the URLLoaderDataFormat class. The three possible values for the `URLLoader.dataFormat` property are as follows:

- `URLLoaderDataFormat.BINARY`—The `URLLoader.data` property will contain binary data stored in a ByteArray object.
- `URLLoaderDataFormat.TEXT`—The `URLLoader.data` property will contain text in a String object.
- `URLLoaderDataFormat.VARIABLES`—The `URLLoader.data` property will contain URL-encoded variables stored in a URLVariables object.

The following code demonstrates how setting the `URLLoader.dataFormat` property to `URLLoaderDataFormat.VARIABLES` allows you to automatically parse loaded data into a URLVariables object:

```
package
{
  import flash.display.Sprite;
  import flash.events.*;
  import flash.net.URLLoader;
  import flash.net.URLLoaderDataFormat;
  import flash.net.URLRequest;

  public class URLLoaderDataFormatExample extends Sprite
  {
    public function URLLoaderDataFormatExample()
    {
      var request:URLRequest = new URLRequest("http://
  www.[yourdomain].com/params.txt");
      var variables:URLLoader = new URLLoader();
      variables.dataFormat = URLLoaderDataFormat.VARIABLES;
      variables.addEventListener(Event.COMPLETE, completeHandler);
      try
      {
        variables.load(request);
      }
      catch (error:Error)
      {
        trace("Unable to load URL: " + error);
      }
    }
    private function completeHandler(event:Event):void
    {
        var loader:URLLoader = URLLoader(event.target);
        trace(loader.data.dayNames);
    }
  }
}
```

NOTE

The default value for `URLLoader.dataFormat` is `URLLoaderDataFormat.TEXT`.

As the following example shows, Loading XML from an external file is the same as loading URLVariables. You can create a URLRequest instance and a URLLoader instance and use them to download a remote XML document. When the file has completely downloaded, the `Event.COMPLETE` event is dispatched and the contents of the external file are converted to an XML instance, which you can parse using XML methods and properties.

```
package
{
  import flash.display.Sprite;
  import flash.errors.*;
  import flash.events.*;
  import flash.net.URLLoader;
  import flash.net.URLRequest;

  public class ExternalDocs extends Sprite
  {
    public function ExternalDocs()
    {
      var request:URLRequest = new URLRequest("http://
  www.[yourdomain].com/data.xml");
      var loader:URLLoader = new URLLoader();
      loader.addEventListener(Event.COMPLETE, completeHandler);
      try
      {
        loader.load(request);
      }
      catch (error:ArgumentError)
      {
        trace("An ArgumentError has occurred.");
      }
      catch (error:SecurityError)
      {
        trace("A SecurityError has occurred.");
      }
    }
    private function completeHandler(event:Event):void
    {
      var dataXML:XML = XML(event.target.data);
      trace(dataXML.toXMLString());
    }
  }
}
```

Communicating with external scripts

In addition to loading external data files, you can also use the URLVariables class to send variables to a server-side script and process the server's response. This is useful, for example, if you are programming a game and want to send the user's score to a server to calculate whether it should be added to the high scores list, or even send a user's login information to a server for validation. A server-side script can process the user name and password, validate it against a database, and return confirmation of whether the user-supplied credentials are valid.

The following snippet creates a URLVariables object named `variables`, which creates a new variable called `name`. Next, a URLRequest object is created that specifies the URL of the server-side script to send the variables to. Then you set the `method` property of the URLRequest object to send the variables as an HTTP `POST` request. To add the URLVariables object to the URL request, you set the `data` property of the URLRequest object to the URLVariables object created earlier. Finally, the URLLoader instance is created and the `URLLoader.load()` method is invoked, which initiates the request.

```
var variables:URLVariables = new URLVariables("name=Franklin");
var request:URLRequest = new URLRequest();
request.url = "http://www.[yourdomain].com/greeting.cfm";
request.method = URLRequestMethod.POST;
request.data = variables;
var loader:URLLoader = new URLLoader();
loader.dataFormat = URLLoaderDataFormat.VARIABLES;
loader.addEventListener(Event.COMPLETE, completeHandler);
try
{
  loader.load(request);
}
catch (error:Error)
{
  trace("Unable to load URL");
}

function completeHandler(event:Event):void
{
  trace(event.target.data.welcomeMessage);
}
```

The following code contains the contents of the ColdFusion® greeting.cfm document used in the previous example:

```
<cfif NOT IsDefined("Form.name") OR Len(Trim(Form.Name)) EQ 0>
  <cfset Form.Name = "Stranger" />
</cfif>
<cfoutput>welcomeMessage=#UrlEncodedFormat("Welcome, " & Form.name)#
</cfoutput>
```

Connecting to other Flash Player instances

The LocalConnection class lets you communicate between different Flash Player instances, such as a SWF in an HTML container or in an embedded or stand-alone player. This allows you to build very versatile applications that can share data between Flash Player instances, such as SWF files running in a web browser or embedded in C# applications.

LocalConnection class

The LocalConnection class lets you develop SWF files that can send instructions to other SWF files without the use of the `fscommand()` method or JavaScript. LocalConnection objects can communicate only among SWF files that are running on the same client computer, but they can run in different applications. For example, a SWF file running in a browser and a SWF file running in a projector can share information, with the projector maintaining local information and the browser-based SWF connecting remotely. (A projector is a SWF file saved in a format that can run as a stand-alone application—that is, the projector doesn't require Flash Player to be installed because it is embedded inside the executable.)

LocalConnection objects created in ActionScript 3.0 can communicate with LocalConnection objects created in ActionScript 1.0 or 2.0. The reverse is also true: LocalConnection objects created in ActionScript 1.0 or 2.0 can communicate with LocalConnection objects created in ActionScript 3.0. Flash Player handles this communication between LocalConnection objects of different versions automatically.

The simplest way to use a LocalConnection object is to allow communication only between LocalConnection objects located in the same domain because that way, you won't have security issues. However, if you need to allow communication between domains, you have several ways to implement security measures. For more information, see the discussion of the `connectionName` parameter in `send()` and the `allowDomain()` and `domain` entries in the *ActionScript 3.0 Language Reference*.

TIP

It is possible to use LocalConnection objects to send and receive data within a single SWF file, but Adobe does not recommended doing so. Instead, you should use shared objects.

There are three ways to add callback methods to your LocalConnection objects:

- Subclass the LocalConnection class and add methods.
- Set the `LocalConnection.client` property to an object that implements the methods.
- Create a dynamic class that extends LocalConnection and dynamically attach methods.

The first way to add callback methods is to extend the LocalConnection class. You define the methods within the custom class instead of dynamically adding them to the LocalConnection instance. This approach is demonstrated in the following code:

```
package
{
  import flash.net.LocalConnection;
  public class CustomLocalConnection extends LocalConnection
  {
    public function CustomLocalConnection(connectionName:String)
    {
      try
      {
        connect(connectionName);
      }
      catch (error:ArgumentError)
      {
        // server already created/connected
      }
    }
    public function onMethod(timeString:String):void
    {
      trace("onMethod called at: " + timeString);
    }
  }
}
```

In order to create a new instance of the DynamicLocalConnection class, you can use the following code:

```
var serverLC:CustomLocalConnection;
serverLC = new CustomLocalConnection("serverName");
```

The second way to add callback methods is to use the `LocalConnection.client` property. This involves creating a custom class and assigning a new instance to the `client` property, as the following code shows:

```
var lc:LocalConnection = new LocalConnection();
lc.client = new CustomClient();
```

The `LocalConnection.client` property indicates the object callback methods that should be invoked. In the previous code, the `client` property was set to a new instance of a custom class, CustomClient. The default value for the `client` property is the current LocalConnection instance. You can use the `client` property if you have two data handlers that have the same set of methods but act differently—for example, in an application where a button in one window toggles the view in a second window.

To create the CustomClient class, you could use the following code:

```
package
{
  public class CustomClient extends Object
  {
    public function onMethod(timeString:String):void
    {
      trace("onMethod called at: " + timeString);
    }
  }
}
```

The third way to add callback methods, creating a dynamic class and dynamically attaching the methods, is very similar to using the LocalConnection class in earlier versions of ActionScript, as the following code shows:

```
import flash.net.LocalConnection;
dynamic class DynamicLocalConnection extends LocalConnection {}
```

Callback methods can be dynamically added to this class by using the following code:

```
var connection:DynamicLocalConnection = new DynamicLocalConnection();
connection.onMethod = this.onMethod;
// Add your code here.
public function onMethod(timeString:String):void
{
  trace("onMethod called at: " + timeString);
}
```

The previous way of adding callback methods is not recommended because the code is not very portable. In addition, using this method of creating local connections would could create performance issues because accessing dynamic properties is dramatically slower than accessing sealed properties.

Sending messages between two Flash Player instances

You use the LocalConnection class to communicate between different instances of Flash Player. For example, you could have multiple Flash Player instances on a web page, or have a Flash Player instance retrieve data from a Flash Player instance in a pop-up window.

The following code defines a local connection object that acts as a server and accepts incoming calls from other Flash Player instances:

```
package
{
  import flash.net.LocalConnection;
  import flash.display.Sprite;
  public class ServerLC extends Sprite
  {
    public function ServerLC()
    {
      var lc:LocalConnection = new LocalConnection();
      lc.client = new CustomClient1();
      try
      {
        lc.connect("conn1");
      }
      catch (error:Error)
      {
        trace("error:: already connected");
      }
    }
  }
}
```

This code first creates a LocalConnection object named `lc` and sets the `client` property to a custom class, CustomClient1. When another Flash Player instance calls a method in this local connection instance, Flash Player looks for that method in the CustomClient1 class.

Whenever a Flash Player instance connects to this SWF file and tries to invoke any method for the specified local connection, the request is sent to the class specified by the `client` property, which is set to the CustomClient1 class:

```
package
{
  import flash.events.*;
  import flash.system.fscommand;
  import flash.utils.Timer;
  public class CustomClient1 extends Object
  {
    public function doMessage(value:String = ""):void
    {
      trace(value);
    }
    public function doQuit():void
    {
      trace("quitting in 5 seconds");
      this.close();
      var quitTimer:Timer = new Timer(5000, 1);
      quitTimer.addEventListener(TimerEvent.TIMER, closeHandler);
    }
    public function closeHandler(event:TimerEvent):void
    {
      fscommand("quit");
    }
  }
}
```

To create a LocalConnection server, call the `LocalConnection.connect()` method and provide a unique connection name. If you already have a connection with the specified name, an `ArgumentError` error is generated, indicating that the connection attempt failed because the object is already connected.

The following snippet demonstrates how to create a new socket connection with the name `conn1`:

```
try
{
  connection.connect("conn1");
}
catch (error:ArgumentError)
{
  trace("Error! Server already exists\n");
}
```

NOTE In earlier versions of ActionScript, the `LocalConnection.connect()` method returns a Boolean value if the connection name has already been used. In ActionScript 3.0, an error is generated if the name has already been used.

Connecting to the primary SWF file from a secondary SWF file requires that you create a new LocalConnection object in the sending LocalConnection object and then call the `LocalConnection.send()` method with the name of the connection and the name of the method to execute. For example, to connect to the LocalConnection object that you created earlier, you use the following code:

```
sendingConnection.send("conn1", "doQuit");
```

This code connects to an existing LocalConnection object with the connection name `conn1` and invokes the `doQuit()` method in the remote SWF file. If you want to send parameters to the remote SWF file, you specify additional arguments after the method name in the `send()` method, as the following snippet shows:

```
sendingConnection.send("conn1", "doMessage", "Hello world");
```

Connecting to SWF documents in different domains

To allow communications only from specific domains, you call the `allowDomain()` or `allowInsecureDomain()` method of the LocalConnection class and pass a list of one or more domains that are allowed to access this LocalConnection object.

In earlier versions of ActionScript, `LocalConnection.allowDomain()` and `LocalConnection.allowInsecureDomain()` were callback methods that had to be implemented by developers and that had to return a Boolean value. In ActionScript 3.0, `LocalConnection.allowDomain()` and `LocalConnection.allowInsecureDomain()` are both built-in methods, which developers can call just like `Security.allowDomain()` and `Security.allowInsecureDomain()`, passing one or more names of domains to be allowed.

There are two special values that you can pass to the `LocalConnection.allowDomain()` and `LocalConnection.allowInsecureDomain()` methods: `*` and `localhost`. The asterisk value `(*)` allows access from all domains. The string `localhost` allows calls to the SWF file from SWF files that are locally installed.

Flash Player 8 introduced security restrictions on local SWF files. A SWF file that is allowed to access the Internet cannot also have access to the local file system. If you specify `localhost`, any local SWF file can access the SWF file. If the `LocalConnection.send()` method attempts to communicate with a SWF file from a security sandbox to which the calling code does not have access, a `securityError` event (`SecurityErrorEvent.SECURITY_ERROR`) is dispatched. To work around this error, you can specify the caller's domain in the receiver's `LocalConnection.allowDomain()` method.

If you implement communication only between SWF files in the same domain, you can specify a `connectionName` parameter that does not begin with an underscore (_) and does not specify a domain name (for example, `myDomain:connectionName`). Use the same string in the `LocalConnection.connect(connectionName)` command.

If you implement communication between SWF files in different domains, you specify a `connectionName` parameter that begins with an underscore (_). Specifying the underscore makes the SWF file with the receiving LocalConnection object more portable between domains. Here are the two possible cases:

- If the string for `connectionName` does not begin with an underscore (_), Flash Player adds a prefix with the superdomain name and a colon (for example, `myDomain:connectionName`). Although this ensures that your connection does not conflict with connections of the same name from other domains, any sending LocalConnection objects must specify this superdomain (for example, `myDomain:connectionName`). If you move the SWF file with the receiving LocalConnection object to another domain, Flash Player changes the prefix to reflect the new superdomain (for example, `anotherDomain:connectionName`). All sending LocalConnection objects have to be manually edited to point to the new superdomain.
- If the string for `connectionName` begins with an underscore (for example, `_connectionName`), Flash Player does not add a prefix to the string. This means the receiving and sending LocalConnection objects will use identical strings for `connectionName`. If the receiving object uses `LocalConnection.allowDomain()` to specify that connections from any domain will be accepted, you can move the SWF file with the receiving LocalConnection object to another domain without altering any sending LocalConnection objects.

Socket connections

There are two different types of socket connections possible in ActionScript 3.0: XML socket connections and binary socket connections. An XML socket lets you connect to a remote server and create a server connection that remains open until explicitly closed. This lets you exchange XML data between a server and client without having to continually open new server connections. Another benefit of using an XML socket server is that the user doesn’t need to explicitly request data. You can send data from the server without requests, and you can send data to every client connected to the XML socket server.

A binary socket connection is similar to an XML socket except that the client and server don’t need to exchange XML packets specifically. Instead, the connection can transfer data as binary information. This allows you to connect to a wide range of services, including mail servers (POP3, SMTP, and IMAP), and news (NNTP) servers.

Socket class

Introduced in ActionScript 3.0, the Socket class enables ActionScript to make socket connections and to read and write raw binary data. It is similar to the XMLSocket class, but does not dictate the format of the received and transmitted data. The Socket class is useful for interoperating with servers that use binary protocols. By using binary socket connections, you can write code that allows interaction with several different Internet protocols, such as POP3, SMTP, IMAP, and NNTP. This in turn enables Flash Player to connect to mail and news servers.

Flash Player can interface with a server by using the binary protocol of that server directly. Some servers use the big-endian byte order, and some use the little-endian byte order. Most servers on the Internet use the big-endian byte order because “network byte order” is big-endian. The little-endian byte order is popular because the Intel x86 architecture uses it. You should use the endian byte order that matches the byte order of the server that is sending or receiving data. All operations that are performed by the IDataInput and IDataOutput interfaces, and the classes that implement those interfaces (ByteArray, Socket, and URLStream), are encoded by default in big-endian format; that is, with the most significant byte first. This is done to match Java and official network byte order. To change whether big-endian or little-endian byte order is used, you can set the `endian` property to `Endian.BIG_ENDIAN` or `Endian.LITTLE_ENDIAN`.

TIP The Socket class inherits all the methods implemented by the IDataInput and IDataOutput interfaces (located in the flash.utils package), and those methods should be used to write to and read from the Socket.

XMLSocket class

ActionScript provides a built-in XMLSocket class, which lets you open a continuous connection with a server. This open connection removes latency issues and is commonly used for real-time applications such as chat applications or multiplayer games. A traditional HTTP-based chat solution frequently polls the server and downloads new messages using an HTTP request. In contrast, an XMLSocket chat solution maintains an open connection to the server, which lets the server immediately send incoming messages without a request from the client.

To create a socket connection, you must create a server-side application to wait for the socket connection request and send a response to the SWF file. This type of server-side application can be written in a programming language such as Java, Python, or Perl. To use the XMLSocket class, the server computer must run a daemon that understands the protocol used by the XMLSocket class. The protocol is described in the following list:

- XML messages are sent over a full-duplex TCP/IP stream socket connection.
- Each XML message is a complete XML document, terminated by a zero (0) byte.
- An unlimited number of XML messages can be sent and received over a single XMLSocket connection.

NOTE

The XMLSocket class cannot tunnel through firewalls automatically because, unlike the Real-Time Messaging Protocol (RTMP), XMLSocket has no HTTP tunneling capability. If you need to use HTTP tunneling, consider using Flash Remoting or Flash Media Server (which supports RTMP) instead.

The following restrictions apply to how and where an XMLSocket object can connect to the server:

- The `XMLSocket.connect()` method can connect only to TCP port numbers greater than or equal to 1024. One consequence of this restriction is that the server daemons that communicate with the `XMLSocket` object must also be assigned to port numbers greater than or equal to 1024. Port numbers below 1024 are often used by system services such as FTP (21), Telnet (23), SMTP (25), HTTP (80), and POP3 (110), so XMLSocket objects are barred from these ports for security reasons. The port number restriction limits the possibility that these resources will be inappropriately accessed and abused.
- The `XMLSocket.connect()` method can connect only to computers in the same domain where the SWF file resides. This restriction does not apply to SWF files running off a local disk. (This restriction is identical to the security rules for `URLLoader.load()`.) To connect to a server daemon running in a domain other than the one where the SWF resides, you can create a security policy file on the server that allows access from specific domains.

NOTE

Setting up a server to communicate with the XMLSocket object can be challenging. If your application does not require real-time interactivity, use the URLLoader class instead of the XMLSocket class.

You can use the `XMLSocket.connect()` and `XMLSocket.send()` methods of the XMLSocket class to transfer XML to and from a server over a socket connection. The `XMLSocket.connect()` method establishes a socket connection with a web server port. The `XMLSocket.send()` method passes an XML object to the server specified in the socket connection.

When you invoke the `XMLSocket.connect()` method, Flash Player opens a TCP/IP connection to the server and keeps that connection open until one of the following occurs:

- The `XMLSocket.close()` method of the XMLSocket class is called.
- No more references to the XMLSocket object exist.
- Flash Player exits.
- The connection is broken (for example, the modem disconnects).

Creating and connecting to a Java XML socket server

The following code demonstrates a simple XMLSocket server written in Java that accepts incoming connections and displays the received messages in the command prompt window. By default, a new server is created on port 8080 of your local machine, although you can specify a different port number when starting your server from the command line.

Create a new text document and add the following code:

```
import java.io.*;
import java.net.*;

class SimpleServer
{
  private static SimpleServer server;
  ServerSocket socket;
  Socket incoming;
  BufferedReader readerIn;
  PrintStream printOut;

  public static void main(String[] args)
  {
    int port = 8080;

    try
    {
      port = Integer.parseInt(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
      // Catch exception and keep going.
    }

    server = new SimpleServer(port);
  }

  private SimpleServer(int port)
  {
    System.out.println(">> Starting SimpleServer");
    try
    {
      socket = new ServerSocket(port);
      incoming = socket.accept();
      readerIn = new BufferedReader(new
  InputStreamReader(incoming.getInputStream()));
      printOut = new PrintStream(incoming.getOutputStream());
      printOut.println("Enter EXIT to exit.\r");
      out("Enter EXIT to exit.\r");
      boolean done = false;
      while (!done)
      {
        String str = readerIn.readLine();
        if (str == null)
        {
          done = true;
        }
        else
        {
          out("Echo: " + str + "\r");
          if(str.trim().equals("EXIT"))
          {
            done = true;
          }
        }
        incoming.close();
      }
    }
    catch (Exception e)
    {
      System.out.println(e);
    }
  }

  private void out(String str)
  {
    printOut.println(str);
    System.out.println(str);
  }
}
```

Save the document to your hard disk as SimpleServer.java and compile it using a Java compiler, which creates a Java class file named SimpleServer.class.

You can start the XMLSocket server by opening a command prompt and typing `java SimpleServer`. The SimpleServer.class file can be located anywhere on your local computer or network; it doesn't need to be placed in the root directory of your web server.

TIP

If you're unable to start the server because the files are not located within the Java classpath, try starting the server with `java -classpath . SimpleServer`.

To connect to the XMLSocket from your ActionScript application, you need to create a new instance of the XMLSocket class, and call the `XMLSocket.connect()` method while passing a host name and port number, as follows:

```
var xmlsock:XMLSocket = new XMLSocket();
xmlsock.connect("127.0.0.1", 8080);
```

A `securityError` (`flash.events.SecurityErrorEvent`) event occurs if a call to `XMLSocket.connect()` attempts to connect either to a server outside the caller's security sandbox or to a port lower than 1024.

Whenever you receive data from the server, the data event (`flash.events.DataEvent.DATA`) is dispatched:

```
xmlsock.addEventListener(DataEvent.DATA, onData);
private function onData(event:DataEvent):void
{
  trace("[" + event.type + "] " + event.data);
}
```

To send data to the XMLSocket server, you use the `XMLSocket.send()` method and pass an XML object or string. Flash Player converts the supplied parameter to a String object and sends the content to the XMLSocket server followed by a zero (0) byte:

```
xmlsock.send(xmlFormattedData);
```

The `XMLSocket.send()` method does not return a value that indicates whether the data was successfully transmitted. If an error occurred while trying to send data, an IOError error is thrown.

TIP

Each message you send to the XML socket server must be terminated by a newline (`\n`) character.

Storing local data

A shared object, sometimes referred to as a "Flash cookie," is a data file that can be created on your computer by the sites that you visit. Shared objects are most often used to enhance your web-browsing experience—for example, by allowing you to personalize the look and feel of a website that you frequently visit. Shared objects, by themselves, can't do anything to or with the data on your computer. More important, shared objects can never access or remember your e-mail address or other personal information—unless you willingly provide such information.

New shared object instances can be created using the static `SharedObject.getLocal()` or `SharedObject.getRemote()` methods. The `getLocal()` method attempts to load a locally persistent shared object that is available only to the current client, whereas the `getRemote()` method attempts to load a remote shared object that can be shared across multiple clients by means of a server, such as Flash Media Server. If the local or remote shared object do not exist, the `getLocal()` and `getRemote()` methods will create a new SharedObject instance.

The following code attempts to load a local shared object named `test`. If this shared object doesn't exist, a new shared object with this name will be created.

```
var so:SharedObject = SharedObject.getLocal("test");
trace("SharedObject is " + so.size + " bytes");
```

If a shared object named test cannot be found, a new one is created with a size of 0 bytes. If the shared object previously existed, its current size (in bytes) is returned.

You can store data in a shared object by assigning values to the data object, as seen in the following example:

```
var so:SharedObject = SharedObject.getLocal("test");
so.data.now = new Date().time;
trace(so.data.now);
trace("SharedObject is " + so.size + " bytes");
```

If there is already a shared object with the name `test` and the parameter `now`, the existing value is overwritten. You can use the `SharedObject.size` property to determine if a shared object already exists, as the following example shows:

```
var so:SharedObject = SharedObject.getLocal("test");
if (so.size == 0)
{
  // Shared object doesn't exist.
  trace("created...");
  so.data.now = new Date().time;
}
trace(so.data.now);
trace("SharedObject is " + so.size + " bytes");
```

The previous code uses the `size` parameter to determine if the shared object instance with the specified name already exists. If you test the following code, you'll notice that each time you run the code, the shared object is recreated. In order for a shared object to be saved to the user's hard drive, you must explicitly call the `SharedObject.flush()` method, as the following example shows:

```
var so:SharedObject = SharedObject.getLocal("test");
if (so.size == 0)
{
  // Shared object doesn't exist.
  trace("created...");
  so.data.now = new Date().time;
}
trace(so.data.now);
trace("SharedObject is " + so.size + " bytes");
so.flush();
```

When using the `flush()` method to write shared objects to a user's hard drive, you should be careful to check whether the user has explicitly disabled local storage using the Flash Player Settings Manager (http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager07.html), as shown in the following example:

```
var so:SharedObject = SharedObject.getLocal("test");
trace("Current SharedObject size is " + so.size + " bytes.");
so.flush();
```

Values can be retrieved from a shared object by specifying the property's name in the shared object's `data` property. For example, if you run the following code, Flash Player will display how many minutes ago the SharedObject instance was created:

```
var so:SharedObject = SharedObject.getLocal("test");
if (so.size == 0)
{
  // Shared object doesn't exist.
  trace("created...");
  so.data.now = new Date().time;
}
var ageMS:Number = new Date().time - so.data.now;
trace("SharedObject was created " + Number(ageMS / 1000 /
  60).toPrecision(2) + " minutes ago");
trace("SharedObject is " + so.size + " bytes");
so.flush();
```

The first time the previous code is run, a new SharedObject instance named `test` will be created and have an initial size of 0 bytes. Because the initial size is 0 bytes, the `if` statement evaluates to `true` and a new property named `now` is added to the local shared object. The shared object's age is calculated by subtracting the value of the `now` property from the current time. Each subsequent time the previous code is run, the size of the shared object should be greater than 0, and the code will trace how many minutes ago the shared object was created.

Displaying contents of a shared object

Values are stored in shared objects within the `data` property. You can loop over each value within a shared object instance by using a `for..in` loop, as the following example shows:

```
var so:SharedObject = SharedObject.getLocal("test");
so.data.hello = "world";
so.data.foo = "bar";
so.data.timezone = new Date().timezoneOffset;
for (var i:String in so.data)
```

```
{
  trace(i + ":\t" + so.data[i]);
}
```

Creating a secure SharedObject

When you create either a local or remote SharedObject using `getLocal()` or `getRemote()`, there is an optional parameter named `secure` that determines whether access to this shared object is restricted to SWF files that are delivered over an HTTPS connection. If this parameter is set to `true` and your SWF file is delivered over HTTPS, Flash Player creates a new secure shared object or gets a reference to an existing secure shared object. This secure shared object can be read from or written to only by SWF files delivered over HTTPS that call `SharedObject.getLocal()` with the secure parameter set to `true`. If this parameter is set to `false` and your SWF file is delivered over HTTPS, Flash Player creates a new shared object or gets a reference to an existing shared object.

This shared object can be read from or written to by SWF files delivered over non-HTTPS connections. If your SWF file is delivered over a non-HTTPS connection and you try to set this parameter to `true`, the creation of a new shared object (or the access of a previously created secure shared object) fails, an error is thrown, and the shared object is set to `null`. If you attempt to run the following snippet from a non-HTTPS connection, the `SharedObject.getLocal()` method will throw an error, as shown in the following snippet:

```
try
{
  var so:SharedObject = SharedObject.getLocal("contactManager", null,
  true);
}
catch (error:Error)
{
  trace("Unable to create SharedObject.");
}
```

Regardless of the value of this parameter, the created shared objects count toward the total amount of disk space allowed for a domain.

Working with file upload and download

The FileReference class lets you add the ability to upload and download files between a client and a server. Users are prompted to select a file to upload or a location for download from a dialog box (such as the Open dialog box on the Windows operating system).

Each FileReference object that you create with ActionScript refers to a single file on the user's hard disk. The object has properties that contain information about the file's size, type, name, creation date, and modification date.

NOTE: The `creator` property is supported on the Macintosh only. All other platforms return `null`.

You can create an instance of the FileReference class in two ways. You can use the `new` operator, as the following code shows:

```
import flash.net.FileReference;
var myFileReference:FileReference = new FileReference();
```

Or you can call the `FileReferenceList.browse()` method, which opens a dialog box on the user's system to prompt the user to select one or more files to upload and then creates an array of FileReference objects if the user selects one or more files successfully. Each FileReference object represents a file selected by the user from the dialog box. A FileReference object does not contain any data in the FileReference properties (such as `name`, `size`, or `modificationDate`) until one of the following happens:

- The `FileReference.browse()` method or `FileReferenceList.browse()` method has been called, and the user has selected a file from the file picker.
- The `FileReference.download()` method has been called, and the user has selected a file from the file picker.

NOTE: When performing a download, only the `FileReference.name` property is populated before the download is complete. After the file has been downloaded, all properties are available.

While calls to the `FileReference.browse()`, `FileReferenceList.browse()`, or `FileReference.download()` method are executing, most players will continue SWF file playback.

FileReference class

The FileReference class allows you to upload and download files between a user's computer and a server. An operating system dialog box prompts the user to select a file to upload or a location for download. Each FileReference object refers to a single file on the user's disk and has properties that contain information about the file's size, type, name, creation date, modification date, and creator.

FileReference instances can be created in two ways:

- When you use the `new` operator with the FileReference constructor, as in the following:

  ```
  var myFileReference:FileReference = new FileReference();
  ```

- When you call `FileReferenceList.browse()`, which creates an array of FileReference objects.

For uploading and downloading operations, a SWF file can access files only within its own domain, including any domains specified by a cross-domain policy file. You need to put a policy file on the file server if the SWF file initiating the upload or download doesn't come from the same domain as the file server.

NOTE

You can only perform one `browse()` or `download()` action at a time, because only one dialog box can be open at any point.

The server script that handles the file upload should expect an HTTP `POST` request with the following elements:

- `Content-Type` with a value of `multipart/form-data`.
- `Content-Disposition` with a `name` attribute set to "`Filedata`" and a `filename` attribute set to the name of the original file. You can specify a custom `name` attribute by passing a value for the `uploadDataFieldName` parameter in the `FileReference.upload()` method.
- The binary contents of the file.

Here is a sample HTTP `POST` request:

```
POST /handler.asp HTTP/1.1
Accept: text/*
Content-Type: multipart/form-data;
boundary=----------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
User-Agent: Shockwave Flash
Host: www.mydomain.com
Content-Length: 421
Connection: Keep-Alive
Cache-Control: no-cache
```

```
------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
Content-Disposition: form-data; name="Filename"

sushi.jpg
------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
Content-Disposition: form-data; name="Filedata"; filename="sushi.jpg"
Content-Type: application/octet-stream

Test File
------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
Content-Disposition: form-data; name="Upload"

Submit Query
------------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
(actual file data,,,)
```

The following sample HTTP `POST` request sends three `POST` variables: `api_sig`, `api_key`, and `auth_token`, and uses a custom upload data field name value of `"photo"`:

```
POST /handler.asp HTTP/1.1
Accept: text/*
Content-Type: multipart/form-data;
boundary=----------Ij5ae0ae0KM7GI3KM7ei4cH2ei4gL6
User-Agent: Shockwave Flash
Host: www.mydomain.com
Content-Length: 421
Connection: Keep-Alive
Cache-Control: no-cache

------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="Filename"

sushi.jpg
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="api_sig"

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="api_key"

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="auth_token"

XXXXXXXXXXXXXXXXXXXXXXX
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="photo"; filename="sushi.jpg"
Content-Type: application/octet-stream
```

```
(actual file data,,,)
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7
Content-Disposition: form-data; name="Upload"

Submit Query
------------Ij5GI3GI3ei4GI3ei4KM7GI3KM7KM7--
```

Uploading files to a server

To upload files to a server, first call the `browse()` method to allow a user to select one or more files. Next, when the `FileReference.upload()` method is called, the selected file will be transferred to the server. If the user selected multiple files using the `FileReferenceList.browse()` method, Flash Player creates an array of selected files called `FileReferenceList.fileList`. You can then use the `FileReference.upload()` method to upload each file individually.

NOTE: Using the `FileReference.browse()` method allows you to upload single files only. To allow a user to upload multiple files, you must use the `FileReferenceList.browse()` method.

By default, the OS file picker dialog box allows users to pick any file type from the local computer, although developers can specify one or more custom file type filters by using the FileFilter class and passing an array of file filter instances to the `browse()` method:

```
var imageTypes:FileFilter = new FileFilter("Images (*.jpg, *.jpeg, *.gif,
  *.png)", "*.jpg; *.jpeg; *.gif; *.png");
var textTypes:FileFilter = new FileFilter("Text Files (*.txt, *.rtf)",
  "*.txt; *.rtf");
var allTypes:Array = new Array(imageTypes, textTypes);
var fileRef:FileReference = new FileReference();
fileRef.browse(allTypes);
```

When the user has selected the files and clicked the Open button in the OS file picker, the `Event.SELECT` event is dispatched. If the `FileReference.browse()` method was used to select a file to upload, the following code is needed to send the file to a web server:

```
var fileRef:FileReference = new FileReference();
fileRef.addEventListener(Event.SELECT, selectHandler);
fileRef.addEventListener(Event.COMPLETE, completeHandler);
try
{
  var success:Boolean = fileRef.browse();
}
catch (error:Error)
{
  trace("Unable to browse for files.");
}
function selectHandler(event:Event):void
```

```
{
  var request:URLRequest = new URLRequest("http://www.[yourdomain].com/
  fileUploadScript.cfm")
  try
  {
    fileRef.upload(request);
  }
  catch (error:Error)
  {
    trace("Unable to upload file.");
  }
}
function completeHandler(event:Event):void
{
  trace("uploaded");
}
```

TIP You can send data to the server with the `FileReference.upload()` method by using the `URLRequest.method` and `URLRequest.data` properties to send variables using the `POST` or `GET` methods.

When you attempt to upload a file using the `FileReference.upload()` method, any of the following events may be dispatched:

- `Event.OPEN`: Dispatched when an upload operation starts.
- `ProgressEvent.PROGRESS`: Dispatched periodically during the file upload operation.
- `Event.COMPLETE`: Dispatched when the file upload operation completes successfully.
- `SecurityErrorEvent.SECURITY_ERROR`: Dispatched when an upload fails because of a security violation.
- `HTTPStatusEvent.HTTP_STATUS`: Dispatched when an upload fails because of an HTTP error.
- `IOErrorEvent.IO_ERROR`: Dispatched if the upload fails because of any of the following reasons:
  - An input/output error occurred while Flash Player is reading, writing, or transmitting the file.
  - The SWF tried to upload a file to a server that requires authentication (such as a user name and password). During upload, Flash Player does not provide a means for users to enter passwords.

- The `url` parameter contains an invalid protocol. The `FileReference.upload()` method must use either HTTP or HTTPS.

TIP

Flash Player does not offer complete support for servers that require authentication. Only SWF files that are running in a browser using the browser plug-in or ActiveX control can provide a dialog box to prompt the user to enter a user name and password for authentication, and then only for downloads. For uploads using the plug-in or ActiveX control or upload/download using either the stand-alone or external player, the file transfer fails.

If you create a server script in ColdFusion to accept a file upload from Flash Player, you can use code similar to the following:

```
<cffile action="upload" filefield="Filedata" destination="#ExpandPath('./
  ')#" nameconflict="OVERWRITE" />
```

This ColdFusion code uploads the file sent by Flash Player and saves it to the same directory as the ColdFusion template, overwriting any file with the same name. The previous code shows the bare minimum amount of code necessary to accept a file upload; this script should not be used in a production environment. Ideally, you should add data validation to ensure that users upload only accepted file types, such as an image instead of a potentially dangerous server-side script.

The following code demonstrates file uploads using PHP, and it includes data validation. The script limits the number of uploaded files in the upload directory to 10, ensures that the file is less than 200 KB, and permits only JPEG, GIF, or PNG files to be uploaded and saved to the file system.

```
<?php
$MAXIMUM_FILESIZE = 1024 * 200; // 200KB
$MAXIMUM_FILE_COUNT = 10; // keep maximum 10 files on server
echo exif_imagetype($_FILES['Filedata']);
if ($_FILES['Filedata']['size'] <= $MAXIMUM_FILESIZE)
{
  move_uploaded_file($_FILES['Filedata']['tmp_name'], "./temporary/
  ".$_FILES['Filedata']['name']);
  $type = exif_imagetype("./temporary/".$_FILES['Filedata']['name']);
  if ($type == 1 || $type == 2 || $type == 3)
  {
    rename("./temporary/".$_FILES['Filedata']['name'], "./images/
  ".$_FILES['Filedata']['name']);
  }
  else
  {
    unlink("./temporary/".$_FILES['Filedata']['name']);
  }
}
$directory = opendir('./images/');
```

```
$files = array();
while ($file = readdir($directory))
{
  array_push($files, array('./images/'.$file, filectime('./images/
  '.$file)));
}
usort($files, sorter);
if (count($files) > $MAXIMUM_FILE_COUNT)
{
  $files_to_delete = array_splice($files, 0, count($files) -
  $MAXIMUM_FILE_COUNT);
  for ($i = 0; $i < count($files_to_delete); $i++)
  {
    unlink($files_to_delete[$i][0]);
  }
}
print_r($files);
closedir($directory);

function sorter($a, $b)
{
  if ($a[1] == $b[1])
  {
    return 0;
  }
  else
  {
    return ($a[1] < $b[1]) ? -1 : 1;
  }
}
?>
```

You can pass additional variables to the upload script using either the `POST` or `GET` request method. To send additional `POST` variables to your upload script, you can use the following code:

```
var fileRef:FileReference = new FileReference();
fileRef.addEventListener(Event.SELECT, selectHandler);
fileRef.addEventListener(Event.COMPLETE, completeHandler);
fileRef.browse();
function selectHandler(event:Event):void
{
  var params:URLVariables = new URLVariables();
  params.date = new Date();
  params.ssid = "94103-1394-2345";
  var request:URLRequest = new URLRequest("http://www.yourdomain.com/
  FileReferenceUpload/fileupload.cfm");
  request.method = URLRequestMethod.POST;
  request.data = params;
  fileRef.upload(request, "Custom1");
}
function completeHandler(event:Event):void
{
  trace("uploaded");
}
```

The previous example creates a new URLVariables object that you pass to the remote server-side script. In previous versions of ActionScript, you could pass variables to the server upload script by passing values in the query string. ActionScript 3.0 allows you to pass variables to the remote script using a URLRequest object, which allows you to pass data using either the `POST` or `GET` method; this, in turn, makes passing larger sets of data easier and cleaner. In order to specify whether the variables are passed using the `GET` or `POST` request method, you can set the `URLRequest.method` property to either `URLRequestMethod.GET` or `URLRequestMethod.POST`, respectively.

ActionScript 3.0 also lets you override the default `Filedata` upload file field name by providing a second parameter to the `upload()` method, as demonstrated in the previous example (which replaced the default value `Filedata` with `Custom1`).

By default, Flash Player will not attempt to send a test upload, although you can override this by passing a value of `true` as the third parameter to the `upload()` method. The purpose of the test upload is to check whether the actual file upload will be successful and that server authentication, if required, will succeed.

NOTE

A test upload occurs only on Windows-based Flash Players at this time.

Downloading files from a server

You can let users download files from a server using the `FileReference.download()` method, which takes two parameters: `request` and `defaultFileName`. The first parameter is the URLRequest object that contains the URL of the file to download. The second parameter is optional—it lets you specify a default filename that appears in the download file dialog box. If you omit the second parameter, `defaultFileName`, the filename from the specified URL is used.

The following code downloads a file named index.xml from the same directory as the SWF document:

```
var request:URLRequest = new URLRequest("index.xml");
var fileRef:FileReference = new FileReference();
fileRef.download(request);
```

To set the default name to currentnews.xml instead of index.xml, specify the `defaultFileName` parameter, as the following snippet shows:

```
var request:URLRequest = new URLRequest("index.xml");
var fileToDownload:FileReference = new FileReference();
fileToDownload.download(request, "currentnews.xml");
```

Renaming a file can be very useful if the server filename was not intuitive or was server-generated. It's also good to explicitly specify the `defaultFileName` parameter when you download a file using a server-side script, instead of downloading the file directly. For example, you need to specify the `defaultFileName` parameter if you have a server-side script that downloads specific files based on URL variables passed to it. Otherwise, the default name of the downloaded file is the name of your server-side script.

Data can be sent to the server using the `download()` method by appending parameters to the URL for the server script to parse. The following ActionScript 3.0 snippet downloads a document based on which parameters are passed to a ColdFusion script:

```
package
{
  import flash.display.Sprite;
  import flash.net.FileReference;
  import flash.net.URLRequest;
  import flash.net.URLRequestMethod;
  import flash.net.URLVariables;

  public class DownloadFileExample extends Sprite
  {
    private var fileToDownload:FileReference;
    public function DownloadFileExample()
    {
      var request:URLRequest = new URLRequest();
      request.url = "http://www.[yourdomain].com/downloadfile.cfm";
      request.method = URLRequestMethod.GET;
      request.data = new URLVariables("id=2");
      fileToDownload = new FileReference();
      try
      {
        fileToDownload.download(request, "file2.txt");
      }
      catch (error:Error)
      {
        trace("Unable to download file.");
      }
    }
  }
}
```

The following code demonstrates the ColdFusion script, download.cfm, that downloads one of two files from the server, depending on the value of a URL variable:

```
<cfparam name="URL.id" default="1" />
<cfswitch expression="#URL.id#">
  <cfcase value="2">
    <cfcontent type="text/plain" file="#ExpandPath('two.txt')#"
  deletefile="No" />
  </cfcase>
  <cfdefaultcase>
    <cfcontent type="text/plain" file="#ExpandPath('one.txt')#"
  deletefile="No" />
  </cfdefaultcase>
</cfswitch>
```

FileReferenceList class

The FileReferenceList class lets the user select one or more files to upload to a server-side script. The file upload is handled by the `FileReference.upload()` method, which must be called on each file that the user selects.

The following code creates two FileFilter objects (`imageFilter` and `textFilter`) and passes them in an array to the `FileReferenceList.browse()` method. This causes the operating system file dialog box to display two possible filters for file types.

```
var imageFilter:FileFilter = new FileFilter("Image Files (*.jpg, *.jpeg,
  *.gif, *.png)", "*.jpg; *.jpeg; *.gif; *.png");
var textFilter:FileFilter = new FileFilter("Text Files (*.txt, *.rtf)",
  "*.txt; *.rtf");
var fileRefList:FileReferenceList = new FileReferenceList();
try
{
  var success:Boolean = fileRefList.browse(new Array(imageFilter,
  textFilter));
}
catch (error:Error)
{
  trace("Unable to browse for files.");
}
```

Allowing the user to select and upload one or more files by using the FileReferenceList class is the same as using `FileReference.browse()` to select files, although the FileReferenceList allows you to select more than one file. Uploading multiple files requires you to upload each of the selected files by using `FileReference.upload()`, as the following code shows:

```
var fileRefList:FileReferenceList = new FileReferenceList();
fileRefList.addEventListener(Event.SELECT, selectHandler);
fileRefList.browse();
function selectHandler(event:Event):void
```

```
{
  var request:URLRequest = new URLRequest("http://www.[yourdomain].com/
  fileUploadScript.cfm");
  var file:FileReference;
  var files:FileReferenceList = FileReferenceList(event.target);
  var selectedFileArray:Array = files.fileList;
  for (var i:uint = 0; i < selectedFileArray.length; i++)
  {
    file = FileReference(selectedFileArray[i]);
    file.addEventListener(Event.COMPLETE, completeHandler);
    try
    {
      file.upload(request);
    }
    catch (error:Error)
    {
      trace("Unable to upload files.");
    }
  }
}
function completeHandler(event:Event):void
{
  trace("uploaded");
}
```

Because the `Event.COMPLETE` event is added to each individual FileReference object in the array, Flash Player calls the `completeHandler()` method when each individual file finishes uploading.

Example: Building a Telnet client

The Telnet example demonstrates techniques for connecting with a remote server and transmitting data using the Socket class. These techniques the example demonstrates are:

- Creating a custom telnet client using the Socket class
- Sending text to the remote server using a ByteArray object
- Handling received data from a remote server

The Telnet application files can be found in the Samples/Telnet folder. The application consists of the following files:

File	Description
TelnetSocket.mxml	The main application file consisting of the MXML user interface.
com/example/programmingas3/Telnet/ Telnet.as	Provides the Telnet client functionality for the application, such as connecting to a remote server, and sending, receiving, and displaying data.

Telnet socket application overview

The main TelnetSocket.mxml file is responsible for creating the user interface (UI) for the entire application.

In addition to the UI, this file also defines two methods, `login()` and `sendCommand()`, to connect the user to the specified server.

The following code lists the ActionScript in the main application file:

```
import com.example.programmingas3.socket.Telnet;
private var telnetClient:Telnet;
private function connect():void
{
  telnetClient = new Telnet(serverName.text, int(portNumber.text), output);
  console.title = "Connecting to " + serverName.text + ":" +
  portNumber.text;
  console.enabled = true;
}
private function sendCommand():void
{
  var ba:ByteArray = new ByteArray();
  ba.writeMultiByte(command.text + "\n", "UTF-8");
  telnetClient.writeBytesToSocket(ba);
  command.text = "";
}
```

The first line of code imports the Telnet class from the custom com.example.programmingas.socket package. The second line of code declares an instance of the Telnet class, `telnetClient`, that will be initialized later by the `connect()` method. Next, the `connect()` method is declared and initializes the `telnetClient` variable declared earlier. This method passes the user-specified telnet server name, telnet server port, and a reference to a TextArea component on the display list, which is used to display the text responses from the socket server. The final two lines of the `connect()` method set the `title` property for the Panel and enable the Panel component, which allows the user to send data to the remote server. The final method in the main application file, `sendCommand()`, is used to send the user's commands to the remote server as a ByteArray object.

Telnet class overview

The Telnet class is responsible for connecting to the remote Telnet server and sending/receiving data.

The Telnet class declares the following private variables:

```
private var serverURL:String;
private var portNumber:int;
private var socket:Socket;
private var ta:TextArea;
private var state:int = 0;
```

The first variable, `serverURL`, contains the user-specified server address to connect to.

The second variable, `portNumber`, is the port number that the Telnet server is currently running on. By default, the Telnet service runs on port 23.

The third variable, `socket`, is a Socket instance that will attempt to connect to the server defined by the `serverURL` and `portNumber` variables.

The fourth variable, `ta`, is a reference to a TextArea component instance on the Stage. This component is used to display responses from the remote Telnet server, or any possible error messages.

The final variable, `state`, is a numeric value that is used to determine which options your Telnet client supports.

As you saw before, the Telnet class's constructor function is called by the `connect()` method in the main application file.

The Telnet constructor takes three parameters: `server`, `port`, and `output`. The `server` and `port` parameters specify the server name and port number where the Telnet server is running. The final parameter, `output`, is a reference to a TextArea component instance on the Stage where server output will be displayed to users.

```
public function Telnet(server:String, port:int, output:TextArea)
{
  serverURL = server;
  portNumber = port;
  ta = output;
  socket = new Socket();
  socket.addEventListener(Event.CONNECT, connectHandler);
  socket.addEventListener(Event.CLOSE, closeHandler);
  socket.addEventListener(ErrorEvent.ERROR, errorHandler);
  socket.addEventListener(IOErrorEvent.IO_ERROR, ioErrorHandler);
  socket.addEventListener(ProgressEvent.SOCKET_DATA, dataHandler);
  Security.loadPolicyFile("http://" + serverURL + "/crossdomain.xml");
  try
  {
    msg("Trying to connect to " + serverURL + ":" + portNumber + "\n");
    socket.connect(serverURL, portNumber);
  }
  catch (error:Error)
  {
    msg(error.message + "\n");
    socket.close();
  }
}
```

Writing data to a socket

To write data to a Socket connection, you call any of the write methods in the Socket class (such as `writeBoolean()`, `writeByte()`, `writeBytes()`, or `writeDouble()`), and then flush the data in the output buffer using the `flush()` method. In the Telnet server, data is written to the socket connection using the `writeBytes()` method which takes the byte array as a parameter and sends it to the output buffer. The `writeBytesToSocket()` method is as follows:

```
public function writeBytesToSocket(ba:ByteArray):void
{
  socket.writeBytes(ba);
  socket.flush();
}
```

This method gets called by the `sendCommand()` method of the main application file.

Displaying messages from the socket server

Whenever a message is received from the socket server, or an event occurs, the custom `msg()` method is called. This method appends a string to the TextArea on the Stage and calls a custom `setScroll()` method, which causes the TextArea component to scroll to the very bottom. The `msg()` method is as follows:

```
private function msg(value:String):void
{
  ta.text += value;
  setScroll();
}
```

If you didn't automatically scroll the contents of the TextArea component, users would need to manually drag the scroll bars on the text area to see the latest response from the server.

Scrolling a TextArea component

The `setScroll()` method contains a single line of ActionScript that scrolls the TextArea component's contents vertically so the user can see the last line of the returned text. The following snippet shows the `setScroll()` method:

```
public function setScroll():void
{
  ta.verticalScrollPosition = ta.maxVerticalScrollPosition;
}
```

This method sets the `verticalScrollPosition` property, which is the line number of the top row of characters that is currently displayed, and sets it to the value of the `maxVerticalScrollPosition` property.

Example: Uploading and downloading files

The FileIO example demonstrates techniques for perform file downloading and uploading in Flash Player. These techniques are:

- Downloading files to a user's computer
- Uploading files from a user's computer to a server
- Cancelling a download in progress
- Cancelling an upload in progress

The FileIO application files are found in the Samples/FileIO folder. The application consists of the following files:

File	Description
FileIO.mxml	The main application file consisting of the MXML user interface.
com/example/programmingas3/fileio/FileDownload.as	A class that includes methods for downloading files from a server.
com/example/programmingas3/fileio/FileUpload.as	A class that includes methods for uploading files to a server.

FileIO application overview

The File IO application contains the user interface that allows a user to upload or download files using Flash Player. The application first defines a couple of custom components, FileUpload and FileDownload, which can be found in the com.example.programmingas3.fileio package. Once each custom component dispatches its `contentComplete` event, the component's `init()` method is called and passes references to a ProgressBar and Button component instance, which allow users to see the file's upload or download progress or cancel the file transfer in progress.

The relevant code from the FileIO.mxml file is as follows:

```
<example:FileUpload id="fileUpload"
  creationComplete="fileUpload.init(uploadProgress, cancelUpload);" />
<example:FileDownload id="fileDownload"
  creationComplete="fileDownload.init(downloadProgress, cancelDownload);"
  />
```

The following code shows the Upload File panel, which contains a progress bar, and two buttons. The first button, `startUpload`, calls the `FileUpload.startUpload()` method, which calls the `FileReference.browse()` method. The following excerpt shows the code for the Upload File panel:

```
<mx:Panel title="Upload File" paddingTop="10" paddingBottom="10"
  paddingLeft="10" paddingRight="10">
  <mx:ProgressBar id="uploadProgress" label="" mode="manual" />
  <mx:ControlBar horizontalAlign="right">
    <mx:Button id="startUpload" label="Upload..."
  click="fileUpload.startUpload();" />
    <mx:Button id="cancelUpload" label="Cancel"
  click="fileUpload.cancelUpload();" enabled="false" />
  </mx:ControlBar>
</mx:Panel>
```

This code places a ProgressBar component instance and two Button component button instances on the Stage. When the user clicks the Upload button (`startUpload`), an operating system dialog box is launched that allows the user to select a file to upload to a remote server. The other button, `cancelUpload`, is disabled by default, although when a user begins a file upload, the button becomes enabled and allows the user to abort the file transfer at any time.

The code for the Download File panel is as follows:

```
<mx:Panel title="Download File" paddingTop="10" paddingBottom="10"
  paddingLeft="10" paddingRight="10">
  <mx:ProgressBar id="downloadProgress" label="" mode="manual" />
  <mx:ControlBar horizontalAlign="right">
    <mx:Button id="startDownload" label="Download..."
  click="fileDownload.startDownload();" />
    <mx:Button id="cancelDownload" label="Cancel"
  click="fileDownload.cancelDownload();" enabled="false" />
  </mx:ControlBar>
</mx:Panel>
```

This code is very similar to the file upload code. When the user clicks the Download button, (`startDownload`), the `FileDownload.startDownload()` method is called, which begins downloading the file specified in the `FileDownload.DOWNLOAD_URL` variable. As the file downloads, the progress bar updates, showing what percentage of the file has downloaded. The user can cancel the download at any point by clicking the `cancelDownload` button, which immediately stops the file download in progress.

Downloading files from a remote server

Downloading of files from a remote server is handled by the flash.net.FileReference class and the custom com.example.programmingas3.fileio.FileDownload class. When the user clicks the Download button, Flash Player begins to download the file specified in the FileDownload class's `DOWNLOAD_URL` variable.

The FileDownload class begins by defining four variables within the com.example.programmingas3.fileio package, the following code shows:

```
  /**
   * Hard-code the URL of file to download to user's computer.
   */
  private const DOWNLOAD_URL:String = "http://www.yourdomain.com/
  file_to_download.zip";

  /**
   * Create a FileReference instance to handle the file download.
   */
  private var fr:FileReference;
```

```
/**
 * Define reference to the download ProgressBar component.
 */
private var pb:ProgressBar;

/**
 * Define reference to the "Cancel" button which will immediately stop
 * the current download in progress.
 */
private var btn:Button;
```

The first variable, `DOWNLOAD_URL`, contains the path to the file, which gets downloaded onto the user's computer when the user clicks the Download button in the main application file.

The second variable, `fr`, is a FileReference object that gets initialized within the `FileDownload.init()` method and will handle the downloading of the remote file to the user's computer.

The last two variables, `pb` and `btn`, contain references to ProgressBar and Button component instances on the Stage, which get initialized by the `FileDownload.init()` method.

Initializing the FileDownload component

The FileDownload component is initialized by calling the `init()` method in the FileDownload class. This method takes two parameters, `pb` and `btn`, which are ProgressBar and Button component instances, respectively.

The code for the `init()` method is as follows:

```
/**
 * Set references to the components, and add listeners for the OPEN,
 * PROGRESS, and COMPLETE events.
 */
public function init(pb:ProgressBar, btn:Button):void
{
  // Set up the references to the progress bar and cancel button,
  // which are passed from the calling script.
  this.pb = pb;
  this.btn = btn;

  fr = new FileReference();
  fr.addEventListener(Event.OPEN, openHandler);
  fr.addEventListener(ProgressEvent.PROGRESS, progressHandler);
  fr.addEventListener(Event.COMPLETE, completeHandler);
}
```

Beginning the file download

When the user clicks the Download Button component instance on the Stage, the `startDownload()` method is to initiate the file download process. The following excerpt shows the `startDownload()` method:

```
/**
 * Begin downloading the file specified in the DOWNLOAD_URL constant.
 */
public function startDownload():void
{
    var request:URLRequest = new URLRequest();
    request.url = DOWNLOAD_URL;
    fr.download(request);
}
```

First, the `startDownload()` method creates a new URLRequest object, and sets the target URL to the value specified by the `DOWNLOAD_URL` variable. Next, the `FileReference.download()` method is called, and the newly created URLRequest object is passed as a parameter. This causes the operating system to display a dialog box on the user's computer prompting them to select a location to save the requested document. Once the user selects a location, the `open` event (`Event.OPEN`) is dispatched and the `openHandler()` method is invoked.

The `openHandler()` method sets the text format for the ProgressBar component's `label` property, and enables the Cancel button, which allows the user to immediately stop the download in progress. The `openHandler()` method is as follows:

```
/**
 * When the OPEN event has dispatched, change the progress bar's label
 * and enable the "Cancel" button, which allows the user to abort the
 * download operation.
 */
private function openHandler(event:Event):void
{
    pb.label = "DOWNLOADING %3%%";
    btn.enabled = true;
}
```

Monitoring a file's download progress

As a file downloads from a remote server to the user's computer, the `progress` event (`ProgressEvent.PROGRESS`) is dispatched at regular intervals. Whenever the `progress` event is dispatched, the `progressHandler()` method is invoked and the ProgressBar component instance on the Stage is updated. The code for the `progressHandler()` method is as follows:

```
/**
 * While the file is downloading, update the progress bar's status.
 */
private function progressHandler(event:ProgressEvent):void
{
  pb.setProgress(event.bytesLoaded, event.bytesTotal);
}
```

The progress event contains two properties, `bytesLoaded` and `bytesTotal`, which are used to update the ProgressBar component on the Stage. This gives the user a sense of how much of the file has already finished downloading and how much remains. The user can abort the file transfer at any time by clicking the Cancel button below the progress bar.

If the file is downloaded successfully, the `complete` event (`Event.COMPLETE`) invokes the `completeHandler()` method, which notifies the user that the file has completed downloading and disables the Cancel button. The code for the `completeHandler()` method is as follows:

```
/**
 * Once the download has completed, change the progress bar's label one
 * last time and disable the "Cancel" button since the download is
 * already completed.
 */
private function completeHandler(event:Event):void
{
  pb.label = "DOWNLOAD COMPLETE";
  btn.enabled = false;
}
```

Cancelling a file download

A user can abort a file transfer and stop any further bytes from being downloaded at any time by clicking the Cancel button on the Stage. The following excerpt shows the code for cancelling a download:

```
/**
 * Cancel the current file download.
 */
public function cancelDownload():void
{
  fr.cancel();
```

```
    pb.label = "DOWNLOAD CANCELLED";
    btn.enabled = false;
  }
```

First, the code stops the file transfer immediately, preventing any further data from downloading. Next, the progress bar's label property is updated to notify the user that the download has been successfully cancelled. Finally, the Cancel button is disabled, which prevents the user from clicking the button again until they attempt to download the file again.

Uploading files to a remote server

The file upload process is very similar to the file download process. The FileUpload class declares the same four variables, as shown in the following code:

```
private const UPLOAD_URL:String = "http://www.yourdomain.com/
  your_upload_script.cfm";
private var fr:FileReference;
private var pb:ProgressBar;
private var btn:Button;
```

Unlike the `FileDownload.DOWNLOAD_URL` variable, the `UPLOAD_URL` variable contains the URL to the server-side script that will upload the file from the user's computer. The remaining three variables behave the same as their counterparts in the FileDownload class.

Initializing the FileUpload component

The FileUpload component contains an `init()` method, which gets called from the main application. This method takes two parameters, `pb` and `btn`, which are references to a ProgressBar and Button component instance on the Stage. Next, the `init()` method initializes the FileReference object defined earlier in the FileUpload class. Finally, the method assigns four event listeners to the FileReference object. The code for the `init()` method is as follows:

```
public function init(pb:ProgressBar, btn:Button):void
{
    this.pb = pb;
    this.btn = btn;

    fr = new FileReference();
    fr.addEventListener(Event.SELECT, selectHandler);
    fr.addEventListener(Event.OPEN, openHandler);
    fr.addEventListener(ProgressEvent.PROGRESS, progressHandler);
    fr.addEventListener(Event.COMPLETE, completeHandler);
}
```

Beginning a file upload

The file upload is initiated when the user clicks on the Upload button on the Stage, which invokes the `FileUpload.startUpload()` method. This method calls the `browse()` method of the FileReference class which causes the operating system to display a system dialog box prompting the user to select a file to upload to the remote server. The following excerpt shows the code for the `startUpload()` method:

```
public function startUpload():void
{
    fr.browse();
}
```

Once the user selects a file to upload, the `select` event (`Event.SELECT`) is dispatched, causing the `selectHandler()` method to be invoked. The `selectHandler()` method creates a new URLRequest object and sets the `URLRequest.url` property to the value of the `UPLOAD_URL` constant defined earlier in the code. Finally, the FileReference object uploads the selected file to the specified server-side script. The code for the `selectHandler()` method is as follows:

```
private function selectHandler(event:Event):void
{
    var request:URLRequest = new URLRequest();
    request.url = UPLOAD_URL;
    fr.upload(request);
}
```

The remaining code in the FileUpload class is the same as the code defined in the FileDownload class. If a user wishes to terminate the upload at any point, they can click the Cancel button, which sets the label on the progress bar and stops the file transfer immediately. The progress bar gets updated whenever the `progress` event (`ProgressEvent.PROGRESS`) is dispatched. Similarly, once the upload has completed, the progress bar is updated to notify the user that the file has uploaded successfully. The Cancel button is then disabled until the user begins a new file transfer.

CHAPTER 15

Working with Geometry

15

The flash.geom package contains classes that define geometric objects such as points, rectangles, and transformation matrixes. You use these classes to define the properties of objects that are used in other classes.

Contents

Using Point objects

A Point object defines a Cartesian pair of coordinates. It represents location in a two-dimensional coordinate system, where *x* represents the horizontal axis and *y* represents the vertical axis.

To define a Point object, you set its `x` and `y` properties, as follows:

```
import flash.geom.*;
var pt1:Point = new Point(10, 20); // x == 10; y == 20
var pt2:Point = new Point();
pt2.x = 10;
pt2.y = 20;
```

Finding the distance between two points

You can use the `distance()` method of the Point class to find the distance between two points in a coordinate space. For example, the following code finds the distance between the registration points of two display objects, `circle1` and `circle2`, in the same display object container:

```
import flash.geom.*;
var pt1:Point = new Point(circle1.x, circle1.y);
var pt2:Point = new Point(circle2.x, circle2.y);
var distance:Number = Point.distance(pt1, pt2);
```

Translating coordinate spaces

If two display objects are in different display object containers, they may be in different coordinate spaces. You can use the `localToGlobal()` method of the DisplayObject class to translate the coordinates to the same (global) coordinate space, that of the Stage. For example, the following code finds the distance between the registration points of two display objects, `circle1` and `circle2`, in the different display object containers:

```
import flash.geom.*;
var pt1:Point = new Point(circle1.x, circle1.y);
pt1 = circle1.localToGlobal(pt1);
var pt2:Point = new Point(circle1.x, circle1.y);
pt2 = circle2.localToGlobal(pt2);
var distance:Number = Point.distance(pt1, pt2);
```

Similarly, to find the distance of the registration point of a display object named `target` from a specific point on the Stage, you can use the `localToGlobal()` method of the DisplayObject class:

```
import flash.geom.*;
var stageCenter:Point = new Point();
stageCenter.x = this.stage.stageWidth / 2;
stageCenter.y = this.stage.stageHeight / 2;
var targetCenter:Point = new Point(target.x, target.y);
targetCenter = target.localToGlobal(targetCenter);
var distance:Number = Point.distance(stageCenter, targetCenter);
```

Moving a display object by a specified angle and distance

You can use the `polar()` method of the Point class to move a display object a specific distance by a specific angle. For example, the following code moves the `myDisplayObject` object 100 pixels by 60 degrees:

```
import flash.geom.*;
var distance:Number = 100;
var angle:Number = 2 * Math.PI * (90 / 360);
var translatePoint:Point = Point.polar(distance, angle);
myDisplayObject.x += translatePoint.x;
myDisplayObject.y += translatePoint.y;
```

Other uses of the Point class

You can use Point objects with the following methods and properties:

Class	Methods or properties	Description

DisplayObjectContainer	`areInaccessibleObjectsUnderPoint()`
	`getObjectsUnderPoint()`
		Used to return a list of objects under a point in a display object container.

BitmapData	`hitTest()`	Used to define the pixel in the BitmapData object as well as the point that you are checking for a hit.

BitmapData	`applyFilter()`
	`copyChannel()`
	`merge()`
	`paletteMap()`
	`pixelDissolve()`
	`threshold()`
		Used to define the positions of rectangles that define the operations.

Matrix	`deltaTransformPoint()`
	`transformPoint()`
		Used to define points for which you want to apply a transformation.

Rectangle	`bottomRight`
	`size`
	`topLeft`
		Used to define these properties.

Using Rectangle objects

A Rectangle object defines a rectangular area. A Rectangle object has a position, defined by the *x* and *y* coordinates of its top-left corner, a `width` property, and a `height` property. You can define these properties for a new Rectangle object by invoking the `Rectangle()` constructor function, as follows:

```
import flash.geom.Rectangle;
var rx:Number = 0;
var ry:Number = 0;
var rwidth:Number = 100;
var rheight:Number = 50;
var rect1:Rectangle = new Rectangle(rx, ry, rwidth, rheight);
```

Resizing and repositioning Rectangle objects

There are a number of ways to resize and reposition Rectangle objects.

You can directly reposition the Rectangle object by changing its `x` and `y` properties. This has no effect on the width or height of the Rectangle object.

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.x = 20;
rect1.y = 30;
trace(rect1); // (x=20, y=30, w=100, h=50)
```

As the following code shows, if you change the `left` or `top` property of a Rectangle object, it is also repositioned, with its `x` and `y` properties matching the `left` and `top` properties, respectively. However, the position of the bottom-left corner of the Rectangle object does not change, so it is resized.

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.left = 20;
rect1.top = 30;
trace(rect1); // (x=30, y=20, w=70, h=30)
```

Similarly, as the following example shows, if you change the `bottom` or `right` property of a Rectangle object, the position of its top-left corner does not change, so it is resized accordingly:

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.right = 60;
trect1.bottom = 20;
trace(rect1); // (x=0, y=0, w=60, h=20)
```

You can also reposition a Rectangle object by using the `offset()` method, as follows:

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.offset(20, 30);
trace(rect1); // (x=20, y=30, w=100, h=50)
```

The `offsetPt()` method works similarly, except that it takes a Point object as its parameter, rather than *x* and *y* offset values.

You can also resize a Rectangle object by using the `inflate()` method, which includes two parameters, `dx` and `dy`. The `dx` parameter represents the number of pixels that the left and right sides of the rectangle will move from the center, and the `dy` parameter represents the number of pixels that the top and bottom of the rectangle will move from the center:

```
import flash.geom.Rectangle;
var x1:Number = 0;
var y1:Number = 0;
var width1:Number = 100;
var height1:Number = 50;
var rect1:Rectangle = new Rectangle(x1, y1, width1, height1);
trace(rect1) // (x=0, y=0, w=100, h=50)
rect1.inflate(6,4);
trace(rect1); // (x=-6, y=-4, w=112, h=58)
```

The `inflatePt()` method works similarly, except that it takes a Point object as its parameter, rather than `dx` and `dy` values.

Finding unions and intersections of Rectangle objects

You use the `union()` method to find the rectangular region formed by the boundaries of two rectangles:

```
import flash.display.*;
import flash.geom.Rectangle;
var rect1:Rectangle = new Rectangle(0, 0, 100, 100);
trace(rect1); // (x=0, y=0, w=100, h=100)
var rect2:Rectangle = new Rectangle(120, 60, 100, 100);
trace(rect2); // (x=120, y=60, w=100, h=100)
trace(rect1.union(rect2)); // (x=0, y=0, w=220, h=160)
```

You use the `intersection()` method to find the rectangular region formed by the overlapping region of two rectangles:

```
import flash.display.*;
import flash.geom.Rectangle;
var rect1:Rectangle = new Rectangle(0, 0, 100, 100);
trace(rect1); // (x=0, y=0, w=100, h=100)
var rect2:Rectangle = new Rectangle(80, 60, 100, 100);
trace(rect2); // (x=120, y=60, w=100, h=100)
trace(rect1.intersection(rect2)); // (x=80, y=60, w=20, h=40)
```

You use the `intersects()` method to find out whether two rectangles intersect. You can also use the `intersects()` method to find out whether a display object is in a certain region of the Stage. For example, in the following code, assume that the coordinate space of the display object container that contains the `circle` object is the same as that of the Stage. The example shows how to use the `intersects()` method to determine if a display object, `circle`, intersects specified regions of the Stage, defined by the `target1` and `target2` Rectangle objects:

```
import flash.display.*;
import flash.geom.Rectangle;
var circle:Shape = new Shape();
circle.graphics.lineStyle(2, 0xFF0000);
circle.graphics.drawCircle(250, 250, 100);
addChild(circle);
var circleBounds:Rectangle = circle.getBounds(stage);
var target1:Rectangle = new Rectangle(0, 0, 100, 100);
trace(circleBounds.intersects(target1)); // false
var target2:Rectangle = new Rectangle(0, 0, 300, 300);
trace(circleBounds.intersects(target2)); // true
```

Similarly, you can use the `intersects()` method to find out whether the bounding rectangles of two display objects overlap. You can use the `getRect()` method of the DisplayObject class to include any additional space that the strokes of a display object may add to a bounding region.

Other uses of Rectangle objects

Rectangle objects are used in the following methods and properties:

Class	Methods or properties	Description
BitmapData	`applyFilter()`, `colorTransform()`, `copyChannel()`, `copyPixels()`, `draw()`, `fillRect()`, `generateFilterRect()`, `getColorBoundsRect()`, `getPixels()`, `merge()`, `paletteMap()`, `pixelDissolve()`, `setPixels()`, and `threshold()`	Used as the type for some parameters to define a region of the BitmapData object.
DisplayObject	`getBounds()`, `getRect()`, `scrollRect`, `scale9Grid`	Used as the data type for the property or the data type returned.
PrintJob	`addPage()`	Used to define the `printArea` parameter.
Sprite	`startDrag()`	Used to define the `bounds` parameter.
TextField	`getCharBoundaries()`	Used as the return value type.
Transform	`pixelBounds`	Used as the data type.

Using Matrix objects

The Matrix class represents a transformation matrix that determines how to map points from one coordinate space to another. You can perform various graphical transformations on a display object by setting the properties of a Matrix object, applying that Matrix object to the `matrix` property of a Transform object, and then applying that Transform object as the `transform` property of the display object. These transformation functions include translation (*x* and *y* repositioning), rotation, scaling, and skewing.

Defining Matrix objects

Although you could define a matrix by directly adjusting the properties (`a`, `b`, `c`, `d`, `tx`, `ty`) of a Matrix object, it is easier to use the `createBox()` method. This method includes parameters that let you directly define the scaling, rotation, and translation effects of the resulting matrix. For example, the following code creates a Matrix object that has the effect of scaling an object horizontally by 2.0, scaling it vertically by 3.0, rotating it by 45 degrees, moving (translating) it 10 pixels to the right, and moving it 20 pixels down:

```
var matrix:Matrix = new Matrix();
var scaleX:Number = 2.0;
var scaleY:Number = 3.0;
var rotation:Number = 2 * Math.PI * (45 / 360);
var tx:Number = 10;
var ty:Number = 20;
matrix.createBox(scaleX, scaleY, rotation, tx, ty);
```

You can also adjust the scaling, rotation, and translation effects of a Matrix object by using the `scale()`, `rotate()`, and `translate()` methods. Note that these methods combine with the values of the existing Matrix object. For instance, the following code sets a Matrix object that scales an object by a factor of 4 and rotates it 60 degrees, since the `scale()` and `rotate()` methods are called twice:

```
var matrix:Matrix = new Matrix();
var rotation:Number = 2 * Math.PI * (30 / 360); // 30°
var scaleFactor:Number = 2;
matrix.scale(scaleFactor, scaleFactor);
matrix.rotate(rotation);
matrix.scale(scaleX, scaleY);
matrix.rotate(rotation);

myDisplayObject.transform.matrix = matrix;
```

To apply a skew transformation to a Matrix object, adjust its `b` or `c` property. Adjusting the `b` property skews the matrix vertically, and adjusting the `c` property skews the matrix horizontally. The following code skews the `myMatrix` Matrix object vertically by a factor of 2:

```
var skewMatrix:Matrix = new Matrix();
skewMatrix.b = Math.tan(2);
myMatrix.concat(skewMatrix);
```

You can apply a Matrix transformation to the `transform` property of a display object. For example, the following code applies a matrix transformation to a display object named `myDisplayObject`:

```
var matrix:Matrix = myDisplayObject.transform.matrix;
var scaleFactor:Number = 2;
var rotation:Number = 2 * Math.PI * (60 / 360); // 60°
matrix.scale(scaleFactor, scaleFactor);
matrix.rotate(rotation);

myDisplayObject.transform.matrix = matrix;
```

The first line sets a Matrix object to the existing transformation matrix used by the `myDisplayObject` display object (the `matrix` property of the `transformation` property of the `myDisplayObject` display object). This way, the Matrix class methods that you call will have a cumulative effect on the display object's existing position, scale, and rotation.

NOTE

The ColorTransform class is also included in the flash.geometry package. This class is used to set the `colorTransform` property of a Transform object. Since it does not apply any sort of geometrical transformation, it is not discussed in this chapter. For more information, see the ColorTransform class in the *ActionScript 3.0 Language Reference*.

Defining a Matrix object for use with a gradient

You use the `beginGradientFill()` and `lineGradientStyle()` methods of the flash.display.Graphics class to define gradients for use in shapes. When you define a gradient, you supply a matrix as one of the parameters of these methods.

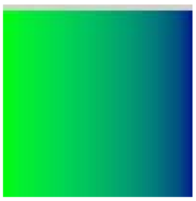
The easiest way to define the matrix is by using the `createGradientBox()` method, which defines a rectangle that is used to define the gradient. You define the scale, rotation, and position of the gradient by using the parameters passed to the `createGradientBox()` method.

For example, consider a gradient with the following characteristics:

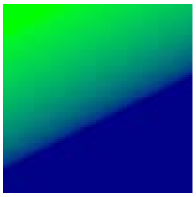
- `GradientType.LINEAR`
- Two colors, green and blue, with the `ratios` array set to `[0, 255]`
- `SpreadMethod.PAD`
- `InterpolationMethod.LINEAR_RGB`

The following examples show gradients in which the `rotation` parameter of the `createGradientBox()` method differs as indicated, but all other settings stay the same:

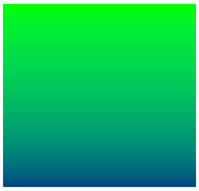
```
width = 100;
height = 100;
rotation = 0;
tx = 0;
ty = 0;
```



```
width = 100;
height = 100;
rotation = Math.PI/4; // 45°
tx = 0;
ty = 0;
```

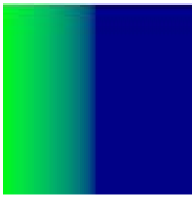


```
width = 100;
height = 100;
rotation = Math.PI/2; // 90°
tx = 0;
ty = 0;
```

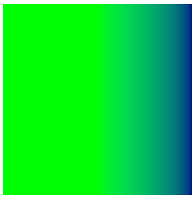


The following examples show the effects on a green-to-blue linear gradient in which the `rotation`, `tx`, and `ty` parameters of the `createGradientBox()` method differ as indicated, but all other settings stay the same:

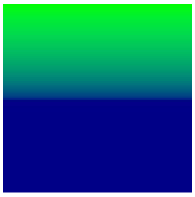
```
width = 50;
height = 100;
rotation = 0;
tx = 0;
ty = 0;
```



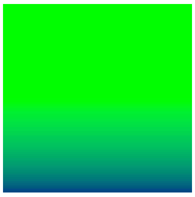
```
width = 50;
height = 100;
rotation = 0
tx = 50;
ty = 0;
```



```
width = 100;
height = 50;
rotation = Math.PI/2; // 90°
tx = 0;
ty = 0;
```

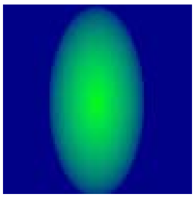


```
width = 100;
height = 50;
rotation = Math.PI/2; // 90°
tx = 0;
ty = 50;
```



The `width`, `height`, `tx`, and `ty` parameters of the `createGradientBox()` method affect the size and position of a *radial* gradient fill as well, as the following example shows:

```
width = 50;
height = 100;
rotation = 0;
tx = 25;
ty = 0;
```



The following code produces the last radial gradient illustrated:

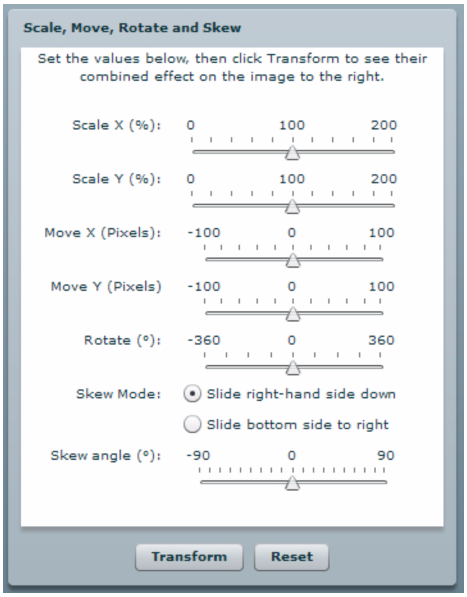
```
import flash.display.Shape;
import flash.display.GradientType;
import flash.geom.Matrix;

var type:String = GradientType.RADIAL;
var colors:Array = [0x00FF00, 0x000088];
var alphas:Array = [1, 1];
var ratios:Array = [0, 255];
var spreadMethod:String = SpreadMethod.PAD;
var interp:String = InterpolationMethod.LINEAR_RGB;
var focalPtRatio:Number = 0;

var matrix:Matrix = new Matrix();
var boxWidth:Number = 50;
var boxHeight:Number = 100;
var boxRotation:Number = Math.PI/2; // 90°
var tx:Number = 25;
var ty:Number = 0;
matrix.createGradientBox(boxWidth, boxHeight, boxRotation, tx, ty);

var square:Shape = new Shape;
square.graphics.beginGradientFill(type,
                colors,
                alphas,
                ratios,
                matrix,
                spreadMethod,
                interp,
                focalPtRatio);
square.graphics.drawRect(0, 0, 100, 100);
addChild(square);
```

Example: Applying a matrix transformation to a display object

The DisplayObjectTransformer sample application shows a number of features of using the Matrix class to transform a display object, including the following:

- Rotating the display object
- Scaling the display object
- Translating (repositioning) the display object
- Skewing the display object

The application provides an interface for adjusting the parameters of the matrix transformation, as follows:



Scale, Move, Rotate and Skew

Set the values below, then click Transform to see their combined effect on the image to the right.

Scale X (%): 0 100 200

Scale Y (%): 0 100 200

Move X (Pixels): -100 0 100

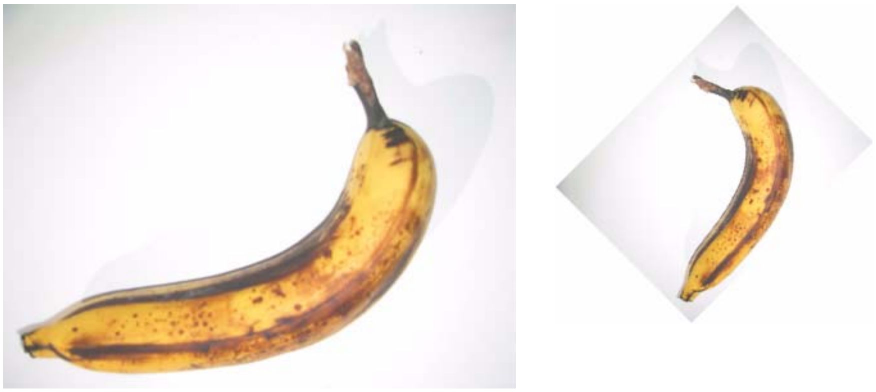
Move Y (Pixels) -100 0 100

Rotate (°): -360 0 360

Skew Mode: [✓ Slide right-hand side down] Slide bottom side to right

Skew angle (°): -90 0 90

When the user clicks the Transform button, the application applies the appropriate transformation.



The original display object, and the display object rotated by -45° and scaled by 50%

The DisplayObjectTransformer application files can be found in the folder Samples/ DisplayObjectTransformer. The application consists of the following files:

File	Description
DisplayObjectTransformer.mxml	The main application file in MXML for Flex.
com/example/programmingas3/geometry/ MatrixTransformer.as	A class that contains methods for applying matrix transformations.
img/	A directory containing sample image files used by the application.

Defining the MatrixTransformer class

The MatrixTransformer class includes static methods that apply geometric transformations of Matrix objects.

The transform() method

The `transform()` method includes parameters for each of the following:

- `sourceMatrix`—The input matrix, which the method transforms
- `xScale` and `yScale`—The *x* and *y* scale factor
- `dx` and `dy`—The *x* and *y* translation amounts, in pixels
- `rotation`—The rotation amount, in degrees
- `skew`—The skew factor, as a percentage
- `skewType`—The direction in which the skew, either `"right"` or `"left"`

The return value is the resulting matrix.

The `transform()` method calls the following static methods of the class:

- `skew()`
- `scale()`
- `translate()`
- `rotate()`

Each returns the source matrix with the applied transformation.

The skew() method

The `skew()` method skews the matrix by adjusting the `b` and `c` properties of the matrix. An optional parameter, `unit`, determines the units used to define the skew angle, and if necessary, the method converts the `angle` value to radians:

```
if (unit == "degrees")
{
  angle = Math.PI * 2 * angle / 360;
}
if (unit == "gradients")
{
  angle = Math.PI * 2 * angle / 100;
}
```

A `skewMatrix` Matrix object is created and adjusted to apply the skew transformation. Initially, it is the identity matrix, as follows:

```
var skewMatrix:Matrix = new Matrix();
```

The `skewSide` parameter determines the side to which the skew is applied. If it is set to `"right"`, the following code sets the `b` property of the matrix:

```
skewMatrix.b = Math.tan(angle);
```

Otherwise, the bottom side is skewed by adjusting the `c` property of the Matrix, as follows:

```
skewMatrix.c = Math.tan(angle);
```

The resulting skew is then applied to the existing matrix by concatenating the two matrixes, as the following example shows:

```
sourceMatrix.concat(skewMatrix);
return sourceMatrix;
```

The scale() method

As the following example shows, the `scale()` method first adjusts the scale factor if it is provided as a percentage, and then uses the `scale()` method of the matrix object:

```
if (percent)
{
  xScale = xScale / 100;
  yScale = yScale / 100;
}
sourceMatrix.scale(xScale, yScale);
return sourceMatrix;
```

The translate() method

The `translate()` method simply applies the `dx` and `dy` translation factors by calling the `translate()` method of the matrix object, as follows:

```
sourceMatrix.translate(dx, dy);
return sourceMatrix;
```

The rotate() method

The `rotate()` method converts the input rotation factor to radians (if it is provided in degrees or gradients), and then calls the `rotate()` method of the matrix object:

```
if (unit == "degrees")
{
  angle = Math.PI * 2 * angle / 360;
}
if (unit == "gradients")
{
  angle = Math.PI * 2 * angle / 100;
}
sourceMatrix.rotate(angle);
return sourceMatrix;
```

Calling the MatrixTransformer.transform() method from the application

The application contains a user interface for getting the transformation parameters from the user. It then passes these, along with the `matrix` property of the `transform` property of the display object, to the `Matrix.transform()` method, as follows:

```
tempMatrix = MatrixTransformer.transform(tempMatrix,
                                    xScaleSlider.value,
                                    yScaleSlider.value,
                                    dxSlider.value,
                                    dySlider.value,
                                    rotationSlider.value,
                                    skewSlider.value,
                                    skewSide );
```

The application then applies the return value to the `matrix` property of the `transform` property of the display object, thereby triggering the transformation:

```
img.content.transform.matrix = tempMatrix;
```

CHAPTER 16

Client System Environment

16

The client system environment is a collection of classes in the flash.system package that allow you to access system-level functionality such as determining which application and security domain a SWF is executing in, determining the capabilities of the user's Flash Player, building multilingual sites using the Input Method Editor (IME), interacting with the Flash Player's container (which could be an HTML page or a container application), or saving information to the user's Clipboard. The flash.system package also includes the IMEConversionMode and SecurityPanel classes. These classes contain static constants that you use with the IME and Security classes, respectively.

This chapter explains how to interact with the user's system. It shows you how to determine what features are supported and how to build multilingual SWF files using the user's installed IME (if available). It also shows typical uses for application domains.

Contents

System class

The System class contains methods and properties that allow you to interact with the user's operating system and retrieve the current memory usage for Flash Player. The methods and properties of the System class also allow you to listen for `imeComposition` events, instruct Flash Player to load external text files using the user's current code page or to load them as Unicode, or set the contents of the user's Clipboard.

Getting data about the user's system at runtime

By checking the `System.totalMemory` property, you can determine the amount of memory (in bytes) that Flash Player is currently using. This property allows you to monitor memory usage and optimize your applications based on how the memory level changes. For example, if a particular visual effect causes a large increase in memory usage, you may want to consider modifying the effect or eliminating it altogether.

The `System.ime` property is a reference to the currently installed Input Method Editor (IME). This property allows you to listen for `imeComposition` events (`flash.events.IMEEvent.IME_COMPOSITION`) by using the `addEventListener()` method.

The third property in the System class is `useCodePage`.When `useCodePage` is set to `true`, Flash Player uses the traditional code page of the operating system that is running the player to load external text files. If you set this property to `false`, you tell Flash Player to interpret the external file as Unicode.

If you set `System.useCodePage` to `true`, remember that the traditional code page of the operating system running the player must include the characters used in your external text file in order for the text to display. For example, if you load an external text file that contains Chinese characters, those characters cannot display on a system that uses the English Windows code page because that code page does not include Chinese characters.

To ensure that users on all platforms can view the external text files that are used in your SWF files, you should encode all external text files as Unicode and leave `System.useCodePage` set to `false` by default. This way, Flash Player 6 and later interprets the text as Unicode.

Saving text to the Clipboard

The System class includes a method called `setClipboard()` that allows Flash Player to set the contents of the user's Clipboard with a specified string. For security reasons, there is no `Security.getClipboard()` method, since such a method could potentially allow malicious sites to access the data last copied to the user's Clipboard.

The following code illustrates how an error message can be copied to the user's Clipboard when a security error occurs. The error message can be useful if the user wants to report a potential bug with an application.

```
private function securityErrorHandler(event:SecurityErrorEvent):void
{
  var errorString:String = "[" + event.type + "] " + event.text;
  trace(errorString);
  System.setClipboard(errorString);
}
```

Capabilities class

The Capabilities class allows developers to determine the environment in which a SWF file is being run. Using various properties of the Capabilities class, you can find out the resolution of the user's system, whether the user's system supports accessibility software, and the language of the user's operating system, as well as the version of the currently installed Flash Player.

By checking the properties in the Capabilities class, you can customize your application to work best with the specific user's environment. For example, by checking the `Capabilities.screenResolutionX` and `Capabilities.screenResolutionY` properties, you can determine the display resolution the user's system is using and decide which video size may be most appropriate. Or you can check the `Capabilities.hasMP3` property to see if the user's system supports MP3 playback before attempting to load an external MP3 file.

The following code uses a regular expression to parse the Flash Player version that the client is using:

```
var versionString:String = Capabilities.version;
var pattern:RegExp = /^(\w*) (\d*),(\d*),(\d*),(\d*)$/;
var result:Object = pattern.exec(versionString);
if (result != null)
{
  trace("input: " + result.input);
  trace("platform: " + result[1]);
  trace("majorVersion: " + result[2]);
  trace("minorVersion: " + result[3]);
  trace("buildNumber: " + result[4]);
  trace("internalBuildNumber: " + result[5]);
}
else
{
  trace("Unable to match RegExp.");
}
```

If you want to send the user's system capabilities to a server-side script so that the information can be stored in a database, you can use the following ActionScript code:

```
var url:String = "log_visitor.cfm";
var request:URLRequest = new URLRequest(url);
request.method = URLRequestMethod.POST;
request.data = new URLVariables(Capabilities.serverString);
var loader:URLLoader = new URLLoader(request);
```

ApplicationDomain class

The purpose of the ApplicationDomain class is to store a table of ActionScript 3.0 definitions. All code in a SWF file is defined to exist in an application domain. You use application domains to partition classes that are in the same security domain. This allows multiple definitions of the same class to exist and also lets children reuse parent definitions.

You can use application domains when loading an external SWF file written in ActionScript 3.0 using the Loader class API. (Note that you cannot use application domains when loading an image or SWF file written in ActionScript 1.0 or ActionScript 2.0.) All ActionScript 3.0 definitions contained in the loaded class are stored in the application domain. When loading the SWF file, you can specify that the file be included in the same application domain as that of the Loader object, by setting the `applicationDomain` parameter of the LoaderContext object to `ApplicationDomain.currentDomain`. By putting the loaded SWF file in the same application domain, you can access its classes directly. This can be useful if you are loading a SWF file that contains embedded media, which you can access via their associated class names, or if you want to access the loaded SWF file's methods, as shown in the following example:

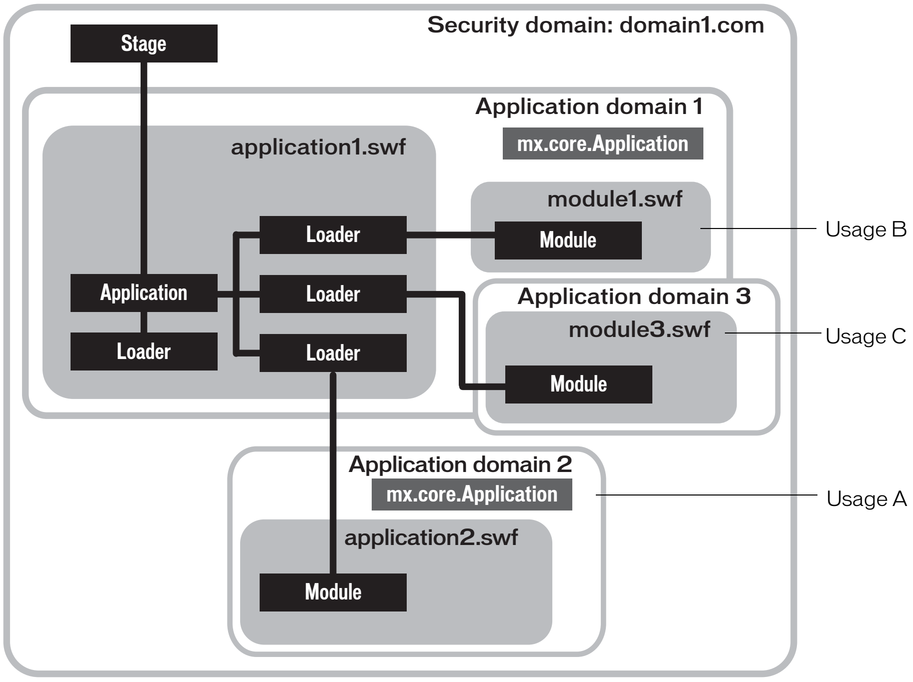
```
package
{
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.events.*;
  import flash.net.URLRequest;
  import flash.system.ApplicationDomain;
  import flash.system.LoaderContext;

  public class ApplicationDomainExample extends Sprite
  {
    private var ldr:Loader;
    public function ApplicationDomainExample()
    {
      ldr = new Loader();
      var req:URLRequest = new URLRequest("Greeter.swf");
      var ldrContext:LoaderContext = new LoaderContext(false,
  ApplicationDomain.currentDomain);
      ldr.contentLoaderInfo.addEventListener(Event.COMPLETE,
  completeHandler);
      ldr.load(req, ldrContext);
    }
    private function completeHandler(event:Event):void
    {
      ApplicationDomain.currentDomain.getDefinition("Greeter");
      var myGreeter:Greeter = Greeter(event.target.content);
      var message:String = myGreeter.welcome("Tommy");
      trace(message); // Hello, Tommy
```

```
        }
    }
}
```

Other things to keep in mind when you work with application domains include the following:

- All code in a SWF file is defined to exist in an application domain. The *current domain* is where your main application runs. The *system domain* contains all application domains, including the current domain, which means that it contains all Flash Player classes.
- All application domains, except the system domain, have an associated parent domain. The parent domain for your main application's application domain is the system domain. Loaded classes are defined only when their parent doesn't already define them. You cannot override a loaded class definition with a newer definition.

The following diagram shows an application that loads content from various SWF files within a single domain, domain1.com. Depending on the content you load, different application domains can be used. The text that follows describes the logic used to set the appropriate application domain for each SWF file in the application.



The main application file is application1.swf. It contains Loader objects that load content from other SWF files. In this scenario, the current domain is Application domain 1. Usage A, usage B, and usage C illustrate different techniques for setting the appropriate application domain for each SWF file in an application.

Usage A: Partition the child SWF file by creating a child of the system domain. In the diagram, application domain 2 is created as a child of the system domain.The application2.swf file is loaded in application domain 2, and its class definitions are thus partitioned from the classes defined in application1.swf.

One use of this technique is to have an old application dynamically loading a newer version of the same application without conflict. There is no conflict because although the same class names are used, they are partitioned into different application domains.

The following code creates an application domain that is a child of the system domain:

```
request.url = "application2.swf";
request.applicationDomain = new ApplicationDomain();
```

Usage B: Add new class definitions to current class definitions. The application domain of module1.swf is set to the current domain (application domain 1). This lets you add to the application's current set of class definitions with new class definitions. This could be used for a run-time shared library of the main application. The loaded SWF is treated as an remote shared library (RSL). Use this technique to load RSLs by a preloader before the application starts.

The following code sets an application domain to the current domain:

```
request.url = "module1.swf";
request.applicationDomain = ApplicationDomain.currentDomain;
```

Usage C: Use the parent's class definitions by creating a new child domain of the current domain. The application domain of module3.swf is a child of the current domain, and the child uses the parent's versions of all classes. One use of this technique might be a module of a multiple-screen rich Internet application (RIA), loaded as a child of the main application, that uses the main application's types. If you can ensure that all classes are always updated to be backward compatible, and that the loading application is always newer than the things it loads, the children will use the parent versions. Having a new application domain also allows you to unload all the class definitions for garbage collection, if you can ensure that you do not continue to have references to the child SWF.

This technique lets loaded modules share the loader's singleton objects and static class members.

The following code creates a new child domain of the current domain:

```
request.url = "module3.swf";
request.applicationDomain = new
  ApplicationDomain(ApplicationDomain.currentDomain);
```

IME class

The IME class lets you manipulate the operating system's Input Method Editor (IME) within Flash Player.

Using ActionScript, you can determine the following:

- If an IME is installed on the user's computer (`Capabilities.hasIME`).
- If the IME is enabled or disabled on the user's computer (`IME.enabled`).
- The conversion mode the current IME is using (`IME.conversionMode`).

You can associate an input text field with a particular IME context. When you switch between input fields, you can also switch the IME between Hiragana (Japanese), full-width numbers, half-width numbers, direct input, and so on.

An IME lets users type non-ASCII text characters in multibyte languages, such as Chinese, Japanese, and Korean.

For more information on working with IMEs, see the documentation for the operating system for which you are developing the application. For additional resources, see the following websites:

- http://www.microsoft.com/globaldev/default.mspx
- http://developer.apple.com/documentation/
- http://java.sun.com/

NOTE

If an IME is not active on the user's computer, calls to IME methods or properties, other than `Capabilities.hasIME`, will fail. Once you manually activate an IME, subsequent ActionScript calls to IME methods and properties will work as expected. For example, if you are using a Japanese IME, you must activate it before you can call any IME method or property.

Checking if an IME is installed and enabled

Before you call any of the IME methods or properties, you should always check to see if the user's computer currently has an IME installed and enabled. The following code illustrates how to check that the user has an IME both installed and active before you call any methods:

```
if (Capabilities.hasIME)
{
  if (IME.enabled)
  {
    trace("IME is installed and enabled.");
  }
  else
  {
    trace("IME is installed but not enabled. Please enable your IME and try
  again.");
  }
}
else
{
  trace("IME is not installed. Please install an IME and try again.");
}
```

The previous code first checks to see if the user has an IME installed using the `Capabilities.hasIME` property. If this property is set to `true`, the code then checks whether the user's IME is currently enabled, using the `IME.enabled` property.

Determining which IME conversion mode is currently enabled

When building multilingual applications, you may need to determine which conversion mode the user currently has active. The following code demonstrates how to check whether the user has an IME installed, and if so, which IME conversion mode is currently active:

```
if (Capabilities.hasIME)
{
  switch (IME.conversionMode)
  {
    case IMEConversionMode.ALPHANUMERIC_FULL:
      tf.text = "Current conversion mode is alphanumeric (full-width).";
      break;
    case IMEConversionMode.ALPHANUMERIC_HALF:
      tf.text = "Current conversion mode is alphanumeric (half-width).";
      break;
    case IMEConversionMode.CHINESE:
      tf.text = "Current conversion mode is Chinese.";
      break;
    case IMEConversionMode.JAPANESE_HIRAGANA:
            tf.text = "Current conversion mode is Japananese Hiragana.";
            break;
        case IMEConversionMode.JAPANESE_KATAKANA_FULL:
            tf.text = "Current conversion mode is Japanese Katakana (full-
    width).";
            break;
        case IMEConversionMode.JAPANESE_KATAKANA_HALF:
            tf.text = "Current conversion mode is Japanese Katakana (half-
    width).";
            break;
        case IMEConversionMode.KOREAN:
            tf.text = "Current conversion mode is Korean.";
            break;
        default:
            tf.text = "Current conversion mode is " + IME.conversionMode + ".";
            break;
    }
}
else
{
    tf.text = "Please install an IME and try again.";
}
```

The previous code first checks to see whether the user has an IME installed. Next it checks which conversion mode the current IME is using by comparing the `IME.conversionMode` property against each of the constants in the IMEConversionMode class.

Setting the IME conversion mode

When you change the conversion mode of the user's IME, you need to make sure that the code is wrapped in a `try..catch` block, because setting a conversion mode using the `conversionMode` property can throw an error if the IME is unable to set the conversion mode. The following code demonstrates how to use a `try..catch` block when setting the `IME.conversionMode` property:

```
var statusText:TextField = new TextField;
statusText.autoSize = TextFieldAutoSize.LEFT;
addChild(statusText);
if (Capabilities.hasIME)
{
    try
    {
        IME.enabled = true;
        IME.conversionMode = IMEConversionMode.KOREAN;
        statusText.text = "Conversion mode is " + IME.conversionMode + ".";
    }
    catch (error:Error)
    {
```

```
    statusText.text = "Unable to set conversion mode.\n" + error.message;
  }
}
```

The previous code first creates a text field, which is used to display a status message to the user. Next, if the IME is installed, the code enables the IME and sets the conversion mode to Korean. If the user's computer does not have a Korean IME installed, an error is thrown by Flash Player and is caught by the `try..catch` block. The `try..catch` block displays the error message in the previously created text field.

Disabling the IME for certain text fields

In some cases, you may want to disable the user's IME while they type characters. For example, if you had a text field that only accepts numeric input, you may not want the IME to come up and slow down data entry.

The following example demonstrates how you can listen for the `FocusEvent.FOCUS_IN` and `FocusEvent.FOCUS_OUT` events and disable the user's IME accordingly:

```
var phoneTxt:TextField = new TextField();
var nameTxt:TextField = new TextField();

phoneTxt.type = TextFieldType.INPUT;
phoneTxt.addEventListener(FocusEvent.FOCUS_IN, focusInHandler);
phoneTxt.addEventListener(FocusEvent.FOCUS_OUT, focusOutHandler);
phoneTxt.restrict = "0-9";
phoneTxt.width = 100;
phoneTxt.height = 18;
phoneTxt.background = true;
phoneTxt.border = true;
addChild(phoneTxt);

nameField.type = TextFieldType.INPUT;
nameField.x = 120;
nameField.width = 100;
nameField.height = 18;
nameField.background = true;
nameField.border = true;
addChild(nameField);

function focusInHandler(event:FocusEvent):void
{
  if (Capabilities.hasIME)
  {
    IME.enabled = false;
  }
}
function focusOutHandler(event:FocusEvent):void
```

```
{
  if (Capabilities.hasIME)
  {
    IME.enabled = true;
  }
}
```

This example creates two input text fields, `phoneTxt` and `nameTxt`, and then adds two event listeners to the `phoneTxt` text field. When the user sets focus to the `phoneTxt` text field, a `FocusEvent.FOCUS_IN` event is dispatched and the IME is disabled. When the `phoneTxt` text field loses focus, the `FocusEvent.FOCUS_OUT` event is dispatched to re-enable the IME.

Listening for IME composition events

IME composition events are dispatched when a composition string is being set. For example, if the user has their IME enabled and active and types a string in Japanese, the `IMEEvent.IME_COMPOSITION` event would dispatch as soon as the user selects the composition string. In order to listen for the `IMEEvent.IME_COMPOSITION` event, you need to add an event listener to the static `ime` property in the System class (`flash.system.System.ime.addEventListener(...)`), as shown in the following example:

```
var inputTxt:TextField;
var outputTxt:TextField;

inputTxt = new TextField();
inputTxt.type = TextFieldType.INPUT;
inputTxt.width = 200;
inputTxt.height = 18;
inputTxt.border = true;
inputTxt.background = true;
addChild(inputTxt);

outputTxt = new TextField();
outputTxt.autoSize = TextFieldAutoSize.LEFT;
outputTxt.y = 20;
addChild(outputTxt);

if (Capabilities.hasIME)
{
  IME.enabled = true;
  try
  {
    IME.conversionMode = IMEConversionMode.JAPANESE_HIRAGANA;
  }
  catch (error:Error)
  {
    outputTxt.text = "Unable to change IME.";
  }
  System.ime.addEventListener(IMEEvent.IME_COMPOSITION,
  imeCompositionHandler);
}
else
{
  outputTxt.text = "Please install IME and try again.";
}

function imeCompositionHandler(event:IMEEvent):void
{
  outputTxt.text = "you typed: " + event.text;
}
```

The previous code creates two text fields and adds them to the display list. The first text field, `inputTxt`, is an input text field that allows the user to enter Japanese text. The second text field, `outputTxt`, is a dynamic text field that displays error messages to the user, or echos the Japanese string that the user types into the `inputTxt` text field.

Example: Detecting system capabilities

The CapabilitiesExplorer example demonstrates how you can use the flash.system.Capabilities class to determine which features the user's Flash Player supports. This example teaches the following techniques:

- Detecting which capabilities the user's Flash Player supports using the Capabilities class
- Using the ExternalInterface class to detect which browser settings the user's browser supports

The CapabilitiesExplorer application files can be found in the folder Samples/CapabilitiesExplorer. This application consists of the following files:

File	Description
CapabilitiesExplorer.mxml	The user interface for the application in MXML for Flex.
com/example/programmingas3/capabilities/CapabilitiesGrabber.as	The class that provides the main functionality of the application, including adding the system Capabilities to an array, sorting the items, and using the ExternalInterface class to retrieve browser capabilities.
capabilities.html	An HTML container that contains the necessary JavaScript to communicate with the External API.

CapabilitiesExplorer overview

The CapabilitiesExplorer.mxml file is responsible for setting up the user interface for the CapabilitiesExplorer application. The user's Flash Player capabilities will be displayed within a DataGrid component instance on the Stage. Their browser capabilities will also be displayed if they are running the application from an HTML container and if the External API is available.

When the main application file's `creationComplete` event is dispatched, the `initApp()` method is invoked. The `initApp()` method calls the `getCapabilities()` method from within the com.example.programmingas3.capabilities.CapabilitiesGrabber class. The code for the `initApp()` method is as follows:

```
private function initApp():void
{
  var dp:Array = CapabilitiesGrabber.getCapabilities();
  capabilitiesGrid.dataProvider = dp;
}
```

The `CapabilitiesGrabber.getCapabilities()` method returns a sorted array of the Flash Player and browser capabilities, which then gets set to the `dataProvider` property of the `capabilitiesGrid` DataGrid component instance on the Stage.

CapabilitiesGrabber class overview

The static `getCapabilities()` method of the CapabilitiesGrabber class adds each property from the flash.system.Capabilities class to an array (`capDP`). It then calls the static `getBrowserObjects()` method in the CapabilitiesGrabber class. The `getBrowserObjects()` method uses the External API to loop over the browser's navigator object, which contains the browser's capabilities. The `getCapabilities()` method is as follows:

```
public static function getCapabilities():Array
{
  var capDP:Array = new Array();
  capDP.push({name:"Capabilities.avHardwareDisable",
  value:Capabilities.avHardwareDisable});
  capDP.push({name:"Capabilities.hasAccessibility",
  value:Capabilities.hasAccessibility});
  capDP.push({name:"Capabilities.hasAudio", value:Capabilities.hasAudio});
  ...
  capDP.push({name:"Capabilities.version", value:Capabilities.version});
  var navArr:Array = CapabilitiesGrabber.getBrowserObjects();
  if (navArr.length > 0)
  {
    capDP = capDP.concat(navArr);
  }
  capDP.sortOn("name", Array.CASEINSENSITIVE);
  return capDP;
}
```

The `getBrowserObjects()` method returns an array of each of the propreties in the browser's navigator object. If this array has a length of one or more items, the array of browser capabilities (`navArr`) is appended to the arrray of Flash Player capabilities (`capDP`), and the entire array is sorted alphabetically. Finally, the sorted array is returned to the main application file, which then populates the data grid. The code for the `getBrowserObjects()` method is as follows:

```
private static function getBrowserObjects():Array
{
  var itemArr:Array = new Array();
  var itemVars:URLVariables;
  if (ExternalInterface.available)
  {
    try
    {
      var tempStr:String = ExternalInterface.call("JS_getBrowserObjects");
      itemVars = new URLVariables(tempStr);
      for (var i:String in itemVars)
      {
        itemArr.push({name:i, value:itemVars[i]});
      }
    }
    catch (error:SecurityError)
    {
      // ignore
    }
  }
  return itemArr;
}
```

If the External API is available in the current user environment, Flash Player calls the JavaScript `JS_getBrowserObjects()` method, which loops over the browser's navigator object and returns a string of URL-encoded values to ActionScript. This string is then converted into a URLVariables object (`itemVars`) and added to the `itemArr` array, which is returned to the calling script.

Communicating with JavaScript

The final piece in building the CapabilitiesExplorer application is writing the necessary JavaScript to loop over each of the items in the browser's navigator object and append a name-value pair to a temporary array. The code for the JavaScript `JS_getBrowserObjects()` method in the container.html file is as follows:

```
<script language="JavaScript">
  function JS_getBrowserObjects()
  {
    // Create an array to hold each of the browser's items.
    var tempArr = new Array();

    // Loop over each item in the browser's navigator object.
    for (var name in navigator)
    {
      var value = navigator[name];

      // If the current value is a string or Boolean object, add it to the
      // array, otherwise ignore the item.
      switch (typeof(value))
      {
        case "string":
        case "boolean":

          // Create a temporary string which will be added to the array.
          // Make sure that we URL-encode the values using JavaScript's
          // escape() function.
          var tempStr = "navigator." + name + "=" + escape(value);
          // Push the URL-encoded name/value pair onto the array.
          tempArr.push(tempStr);
          break;
      }
    }
    // Loop over each item in the browser's screen object.
    for (var name in screen)
    {
      var value = screen[name];

      // If the current value is a number, add it to the array, otherwise
      // ignore the item.
      switch (typeof(value))
      {
        case "number":
          var tempStr = "screen." + name + "=" + escape(value);
          tempArr.push(tempStr);
          break;
      }
    }
    // Return the array as a URL-encoded string of name-value pairs.
    return tempArr.join("&");
  }
</script>
```

The code begins by creating a temporary array that will hold all the name-value pairs in the navigator object. Next, the navigator object is looped over using a `for..in` loop, and the data type of the current value is evaluated to filter out unwanted values. In this application, we are interested only in String or Boolean values, and other data types (such as functions or arrays) are ignored. Each String or Boolean value in the navigator object is appended to the `tempArr` array. Next, the browser's screen object is looped over using a `for..in` loop, and each numeric value is added to the `tempArr` array. Finally, the temporary array is converted into a string using the `Array.join()` method. The array uses an ampersand (&) as a delimiter, which allows ActionScript to easily parse the data using the URLVariables class.

CHAPTER 17

Flash Player Security

17

Security is a key concern of Adobe, users, website owners, and content developers. For this reason, Adobe Flash Player 9 includes a set of security rules and controls to safeguard the user, website owner, and content developer. This chapter discusses how to work with the Flash Player security model when you are developing Flash applications. In this chapter, all SWF files discussed are assumed to be published with ActionScript 3.0 (and thus running in Flash Player 9 or later), unless otherwise noted.

This chapter is intended as an overview of security; it does not try to comprehensively explain all implementation details, usage scenarios, or ramifications for using certain APIs. For a more detailed discussion of Flash Player Security concepts, see the *Flash Player 9 Security* white paper, at www.adobe.com/go/fp9_0_security.

Contents

Flash Player Security overview

Much of Flash Player security is based on the domain of origin for loaded SWF files, media, and other assets. A SWF file from a specific Internet domain, such as www.example.com, can always access all data from that domain. These assets are put in the same security grouping, known as a *security sandbox*. (For more information, see "Security sandboxes" on page 461.)

For example, a SWF file can load SWF files, bitmaps, audio, text files, and any other asset from its own domain. Also, cross-scripting between two SWF files from the same domain is always permitted, as long as both files are written using ActionScript 3.0. *Cross-scripting* is the ability of one SWF file to use ActionScript to access the properties, methods, and objects in another SWF file. Cross-scripting is not supported between SWF files written using ActionScript 3.0 and those using previous versions of ActionScript; however, these files can communicate by using the LocalConnection class. For more information, see "Cross-scripting" on page 470.

The following basic security rules always apply by default:

- Resources in the same security sandbox can always access each other.
- SWF files in a remote sandbox can never access local files and data.

Flash Player considers the following to be individual domains, and sets up individual security sandboxes for each:

- `http://example.com`
- `http://www.example.com`
- `http://store.example.com`
- `https://www.example.com`
- `http://192.0.34.166`

Even if a named domain, such as http://example.com, maps to a specific IP address, such as http://192.0.34.166, Flash Player sets up separate security sandboxes for both.

There are two basic methods that a developer can use to grant a SWF file access to assets from sandboxes other than that of the SWF file:

- The `Security.allowDomain()` method (see "Author (developer) controls" on page 460)
- The cross-domain policy file (see "Website controls (cross-domain policy files)" on page 456)

The ability of a SWF file to cross-script ActionScript 3.0 SWF files from other domains and to load data from other domains is prohibited by default. It can be granted with a call to the `Security.allowDomain()` method in the loaded SWF file. For details, see "Cross-scripting" on page 470.

In the Flash Player security model, there is a distinction between loading *content* and accessing or loading *data*.

- Loading content—*Content* is defined as media, including visual media Flash Player can display, audio, video, or a SWF file that includes displayed media. *Data* is defined as something that is accessible only to ActionScript code. You can load content using classes such as the Loader, Sound, and NetStream classes.
- Accessing content as data or loading data—You can access data in two ways: by extracting data from loaded media content or by directly loading data from an external file (such as an XML file). You can extract data from loaded media by using Bitmap objects, the `BitmapData.draw()` method, the `Sound.id3` property, or the `SoundMixer.computeSpectrum()` method. You can load data using classes such as the URLStream, URLLoader, Socket, and XMLSocket classes.

The Flash Player security model defines different rules for loading content and accessing data. In general, there are fewer restrictions on loading content than on accessing data.

In general, content (SWF files, bitmaps, MP3 files, and videos) can be loaded from anywhere, but if the content is from a domain other than that of the loading SWF file, it will be partitioned in a separate security sandbox.

There are a few barriers to loading content:

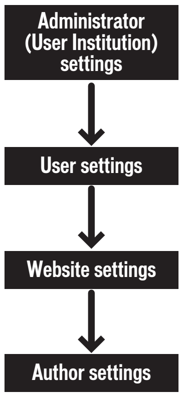
- By default, local SWF files (those loaded from a non-network address, such as a user's hard drive) are classified in the local-with-filesystem sandbox. These files cannot load content from the network. For more information, see "Local sandboxes" on page 462.
- Real-Time Messaging Protocol (RTMP) servers can limit access to content. For more information, see "Content delivered using RTMP servers" on page 470.

If the loaded media is an image, audio, or video, its data, such as pixel data and sound data, cannot be accessed by a SWF file outside its security sandbox, unless the domain of that SWF file has been included in a cross-domain policy file at the origin domain of the media. For details, see "Accessing loaded media as data" on page 474.

Other forms of loaded data include text or XML files, which are loaded with a URLLoader object. Again in this case, to access any data from another security sandbox, permission must be granted by means of a cross-domain policy file at the origin domain. For details, see "Using URLLoader and URLStream" on page 477.

Overview of permission controls

The Flash Player client run-time security model has been designed around resources, which are objects such as SWF files, local data, and Internet URLs. Stakeholders are the parties who own or use those resources. Stakeholders can exercise controls (security settings) over their own resources, and each resource has four stakeholders. Flash Player strictly enforces a hierarchy of authority for these controls, as the following illustration shows:



Hierarchy of security controls

This means, for instance, that if an administrator restricts access to a resource, no other stakeholders can override that restriction.

Administrator, user, and website controls are detailed in the following sections. Author (developer) settings are described in the rest of this chapter.

Administrative user controls

An administrative user of a computer (a user who has logged in with administrative rights) can apply Flash Player security settings that affect all users of the computer. In a nonenterprise environment, such as on a home computer, there is usually one user who also has administrative access. Even in an enterprise environment, individual users may have administrative rights to the computer.

There are two types of administrative user controls:

- The mms.cfg file
- The Global Flash Player Trust directory

The mms.cfg file

On Mac OS X systems, the mms.cfg file is located at /Library/Application Support/Macromedia/mms.cfg. On Microsoft Windows systems, the file is located in the Macromedia Flash Player folder in the system directory (for example, C:\windows\system32\macromed\flash\mms.cfg on a default Windows XP installation).

When Flash Player starts, it reads its security settings from this file, and uses them to limit functionality.

The mms.cfg file includes settings that the administrator uses to perform the following tasks:

- **Data loading**—Restrict the reading of local SWF files, disable file downloading and uploading, and set the storage limit for persistent shared objects.
- **Privacy controls**—Disable microphone and camera access, prevent SWF files from playing windowless content, and prevent SWF files in a domain that does not match the URL displayed in a browser window from accessing persistent shared objects.
- **Flash Player updates**—Set the interval for checking for an updated version of Flash Player, specify the URL to check for Flash Player update information, specify the URL from which to download updated versions of Flash Player, and disable automatic updates of Flash Player entirely.
- **Legacy file support**—Specify whether older-version SWF files should be placed in the local-trusted sandbox.
- **Local file security**—Specify whether local files can be placed in the local-trusted sandbox.
- **Full-screen mode**—Disable full-screen mode.

A SWF file can access some information on capabilities that have been disabled by calling the `Capabilities.avHardwareDisable` and `Capabilities.localFileReadDisable` properties. However, most of the settings in the mms.cfg file cannot be queried from ActionScript.

To enforce application-independent security and privacy policies for a computer, the mms.cfg file should be modified only by system administrators. The mms.cfg file is not for use by application installers. While an installer running with administrative privileges could modify the contents of the mms.cfg file, Adobe considers such usage a violation of the user's trust and urges creators of installers never to modify the mms.cfg file.

The Global Flash Player Trust directory

Administrative users and installer applications can register specified local SWF files as trusted. These SWF files are assigned to the local-trusted sandbox. They can interact with any other SWF files, and they can load data from anywhere, remote or local. Files are designated as trusted in the Global Flash Player Trust directory, which is in the same directory as the mms.cfg file, in the following locations (locations are specific to the current user):

- Windows: system\Macromed\Flash\FlashPlayerTrust

  (for example, C:\windows\system32\Macromed\Flash\FlashPlayerTrust)
- Mac: app support/Macromedia/FlashPlayerTrust

  (for example, /Library/Application Support/Macromedia/FlashPlayerTrust)

The Flash Player Trust directory can contain any number of text files, each of which lists trusted paths, with one path per line. Each path can be an individual SWF file, HTML file, or directory. Comment lines begin with the # symbol. For example, a Flash Player trust configuration file containing the following text grants trusted status to all files in the specified directory and all subdirectories:

```
# Trust files in the following directories:
C:\Documents and Settings\All Users\Documents\SampleApp
```

The paths listed in a trust configuration file should always be local paths or SMB network paths. Any HTTP path in a trust configuration file is ignored; only local files can be trusted.

To avoid conflicts, give each trust configuration file a filename corresponding to the installing application, and use a .cfg file extension.

As a developer distributing a locally run SWF file through an installer application, you can have the installer application add a configuration file to the Global Flash Player Trust directory, granting full privileges to the file that you are distributing. The installer application must be run by a user with administrative rights. Unlike the mms.cfg file, the Global Flash Player Trust directory is included for the purpose of installer applications granting trust permissions. Both administrative users and installer applications can designate trusted local applications using the Global Flash Player Trust directory.

There are also Flash Player Trust directories for individual users (see the next section, "User controls").

User controls

Flash Player provides three different user-level mechanisms for setting permissions: the Settings UI, the Settings Manager, and the User Flash Player Trust directory.

The Settings UI and Settings Manager

The Settings UI is a quick, interactive mechanism for configuring the settings for a specific domain. The Settings Manager presents a more detailed interface and provides the ability to make global changes that affect permissions for many or all domains. Additionally, when a new permission is requested by a SWF file, requiring run-time decisions concerning security or privacy, dialog boxes are displayed in which users can adjust some Flash Player settings.

The Settings Manager and Settings UI provide the following security-related options:

- Camera and microphone settings—The user can control Flash Player access to the camera and microphone on the computer. The user can allow or deny access for all sites or for specific sites. If the user does not specify a setting for all sites or a specific site, a dialog box is displayed when a SWF file attempts to access the camera or microphone, letting the user choose whether or not to allow the SWF file to access the device. The user can also specify the camera or microphone to use, and can set the sensitivity of the microphone.
- Shared object storage settings—The user can select the amount of disk space that a domain can use to store persistent shared objects. The user can make these settings for any numbers of specific domains, and can specify the default setting for new domains. The default limit is 100 KB of disk space. For more information on persistent shared objects, see the SharedObject class in the *ActionScript 3.0 Language Reference*.

NOTE

Any settings made in the mms.cfg file (see "Administrative user controls" on page 452) are not reflected in the Settings Manager.

For details on the Settings Manager, see http://www.adobe.com/go/settingsmanager.

The User Flash Player Trust directory

Users and installer applications can register specified local SWF files as trusted. These SWF files are assigned to the local-trusted sandbox. They can interact with any other SWF files, and they can load data from anywhere, remote or local. A user designates a file as trusted in the User Flash Player Trust directory, which is in same directory as the Flash shared object storage area, in the following locations (locations are specific to the current user):

- Windows: app data\Macromedia\Flash Player\#Security\FlashPlayerTrust

  (for example, C:\Documents and Settings\JohnD\Application Data\Macromedia\Flash Player\#Security\FlashPlayerTrust)
- Mac: app data/Macromedia/Flash Player/#Security/FlashPlayerTrust

  (for example, /Users/JohnD/Library/Preferences/Macromedia/Flash Player/#Security/FlashPlayerTrust)

These settings affect only the current user, not other users who log in to the computer. If a user without administrative rights installs an application in their own portion of the system, the User Flash Player Trust directory lets the installer register the application as trusted for that user.

As a developer distributing a locally run SWF file by way of an installer application, you can have the installer application add a configuration file to the User Flash Player Trust directory, granting full privileges to the file that you are distributing. Even in this situation, the User Flash Player Trust directory file is considered a user control, because a user action (installation) initiates it.

There is also a Global Flash Player Trust directory, used by the administrative user or installers to register an application for all users of a computer (see "Administrative user controls" on page 452).

Website controls (cross-domain policy files)

To make data from a web server available to SWF files from other domains, you can create a cross-domain policy file on your server. A *cross-domain policy file* is an XML file that provides a way for the server to indicate that its data and documents are available to SWF files served from certain domains or from all domains. Any SWF file that is served from a domain specified by the server's policy file is permitted to access data or assets from that server.

Cross-domain policy files affect access to a number of assets, including the following:

- Data in bitmaps, sounds, and videos
- Loading XML and text files
- Access to socket and XML socket connections
- Importing SWF files from other security domains into the security domain of the loading SWF file

Full details are provided in the rest of this chapter.

Policy file syntax

The following example shows a policy file that permits access to SWF files that originate from `*.example.com`, `www.friendOfExample.com` and `192.0.34.166`:

```
<?xml version="1.0"?>
<cross-domain-policy>
  <allow-access-from domain="*.example.com" />
  <allow-access-from domain="www.friendOfExample.com" />
  <allow-access-from domain="192.0.34.166" />
</cross-domain-policy>
```

When a SWF file attempts to access data from another domain, Flash Player automatically attempts to load a policy file from that domain. If the domain of the SWF file that is attempting to access the data is included in the policy file, the data is automatically accessible.

By default, policy files must be named `crossdomain.xml` and must reside in the root directory of the server. However, a SWF file can check for a different name or in a different directory location by calling the `Security.loadPolicyFile()` method. A cross-domain policy file applies only to the directory from which it is loaded and to its child directories. So a policy file in the root directory applies to the whole server, but a policy file loaded from an arbitrary subdirectory applies only to that directory and its subdirectories.

A policy file affects access only to the particular server on which it resides. For example, a policy file located at https://www.adobe.com:8080/crossdomain.xml will apply only to data-loading calls made to www.adobe.com over HTTPS at port 8080.

A cross-domain policy file contains a single `<cross-domain-policy>` tag, which in turn contains zero or more `<allow-access-from>` tags. Each `<allow-access-from>` tag contains an attribute, `domain`, which specifies either an exact IP address, an exact domain, or a wildcard domain (any domain). Wildcard domains are indicated by either a single asterisk (*), which matches all domains and all IP addresses, or an asterisk followed by a suffix, which matches only those domains that end with the specified suffix. Suffixes must begin with a dot. However, wildcard domains with suffixes can match domains that consist of only the suffix without the leading dot. For example, foo.com is considered to be part of *.foo.com. Wildcards are not allowed in IP domain specifications.

If you specify an IP address, access is granted only to SWF files loaded from that IP address using IP syntax (for example, http://65.57.83.12/flashmovie.swf), not those loaded using domain-name syntax. Flash Player does not perform DNS resolution.

You can permit access to documents originating from any domain, as shown in the following example:

```
<?xml version="1.0"?>
<!-- http://www.foo.com/crossdomain.xml -->
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

Each `<allow-access-from>` tag also has the optional `secure` attribute, which defaults to `true`. You can set the attribute to `false` if your policy file is on an HTTPS server, and you want to allow SWF files on a non-HTTPS server to load data from the HTTPS server.

Setting the `secure` attribute to `false` could compromise the security offered by HTTPS. In particular, setting this attribute to `false` opens secure content to snooping and spoofing attacks. Adobe strongly recommends that you not set the `secure` attribute to `false`.

If data to be loaded is on a HTTPS server, but the SWF file loading it is on an HTTP server, Adobe recommends that you move the loading SWF file to an HTTPS server so that you can keep all copies of your secure data under the protection of HTTPS. However, if you decide that you must keep the loading SWF file on an HTTP server, add the `secure="false"` attribute to the `<allow-access-from>` tag, as shown in the following code:

```
<allow-access-from domain="www.example.com" secure="false" />
```

A policy file that contains no `<allow-access-from>` tags has the same effect as not having a policy on a server.

Socket policy files

ActionScript objects instantiate two different kinds of server connections: document-based server connections and socket connections. ActionScript objects like Loader, Sound, URLLoader, and URLStream instantiate document-based server connections, and these each load a file from a URL. ActionScript Socket and XMLSocket objects make socket connections, which operate with streaming data, not loaded documents. Flash Player supports two kinds of policy files: document-based policy files and socket policy files. Document-based connections require document-based policy files, while socket connections require socket policy files.

Flash Player requires that a policy file be transmitted using the same kind of protocol that the attempted connection wishes to use. For example, when you place a policy file on your HTTP server, SWF files from other domains are allowed to load data from it as an HTTP server. However, by not providing a socket policy file at the same server, you are forbidding SWF files from other domains to connect to the server at the socket level. The means by which a socket policy file is retrieved must match the means of connecting.

A policy file served by a socket server has the same syntax as any other policy file, except that it must also specify the ports to which it grants access. When a policy file comes from a port number that is less than 1024, it may grant access to any ports; when a policy file comes from port 1024 or higher, it may only grant access to ports 1024 and higher. The allowed ports are specified in a `to-ports` attribute in the `<allow-access-from>` tag. Single port numbers, port ranges, and wildcards are accepted values.

Here is an example XMLSocket policy file:

```
<cross-domain-policy>
   <allow-access-from domain="*" to-ports="507" />
   <allow-access-from domain="*.example.com" to-ports="507,516" />
   <allow-access-from domain="*.example2.com" to-ports="516-523" />
   <allow-access-from domain="www.example2.com" to-ports="507,516-523" />
   <allow-access-from domain="www.example3.com" to-ports="*" />
</cross-domain-policy>
```

When policy files were first introduced in Flash Player 6, there was no support for socket policy files. Connections to socket servers were authorized by a policy file from the default location of the cross-domain policy file on an HTTP server on port 80 of the same host as the socket server. To make it possible to preserve existing server arrangements, Flash Player 9 still supports this capability. However, Flash Player's default is now to retrieve a socket policy file on the same port as the socket connection. If you wish to use an HTTP-based policy file to authorize a socket connection, you must explicitly request the HTTP policy file using code such as the following:

```
Security.loadPolicyFile("http://socketServerHost.com/crossdomain.xml")
```

In addition, in order to authorize socket connections, an HTTP policy file must come only from the default location of the cross-domain policy file, and not from any other HTTP location. A policy file obtained from an HTTP server implicitly authorizes socket access to all ports 1024 and above; any `to-ports` attributes in an HTTP policy file are ignored.

For more information on socket policy files, see "Connecting to sockets" on page 477.

Preloading policy files

Loading data from a server or connecting to a socket is an asynchronous operation, and Flash Player simply waits for the cross-domain policy file to finish downloading before it begins the main operation. However, extracting pixel data from images or extracting sample data from sounds is a synchronous operation—the cross-domain policy file must load before you can extract data. When you load the media, you need to specify that it check for a cross-domain policy file:

- When using the `Loader.load()` method, set the `checkPolicyFile` property of the `context` parameter, which is a LoaderContext object.
- When embedding an image in a text field using the `<img>` tag, set the `checkPolicyFile` attribute of the `<img>` tag to `"true"`, as in the following: `<img checkPolicyFile = "true" src = "example.jpg">`.
- When using the `Sound.load()` method, set the `checkPolicyFile` property of the `context` parameter, which is a SoundLoaderContext object.
- When using the NetStream class, set the `checkPolicyFile` property of the NetStream object.

When you set one of these parameters, Flash Player first checks for any policy files that it already has downloaded for that domain. Then it considers any pending calls to the `Security.loadPolicyFile()` method to see if they are in scope, and waits for those if they are. Then it looks for the cross-domain policy file in the default location on the server.

Author (developer) controls

The main ActionScript API used to grant security privileges is the `Security.allowDomain()` method, which grant privileges to SWF files in the domains that you specify. In the following example, a SWF file grants access to SWF files served from the www.example.com domain:

```
Security.allowDomain("www.example.com")
```

This method grants permissions for the following:

- Cross-scripting between SWF files (see "Cross-scripting" on page 470)
- Display list access (see "Traversing the display list" on page 473)
- Event detection (see "Event security" on page 473)
- Full access to properties and methods of the Stage object (see "Stage security" on page 472)

The primary purpose of calling the `Security.allowDomain()` method is to grant permission for SWF files in an outside domain to script the SWF file calling the `Security.allowDomain()` method. For more information, see "Cross-scripting" on page 470.

Specifying an IP address as a parameter to the `Security.allowDomain()` method does not permit access by all parties that originate at the specified IP address. Instead, it permits access only by a party that contains the specified IP address as its URL, rather than a domain name that maps to that IP address. For example, if the domain name www.example.com maps to the IP address 192.0.34.166, a call to `Security.allowDomain("192.0.34.166")` does not grant access to www.example.com.

You can pass the `"*"` wildcard to the `Security.allowDomain()` method to allow access from all domains. Because it grants permission for SWF files from *all* domains to script the calling SWF file, use the `"*"` wildcard with care.

ActionScript includes a second permission API, called `Security.allowInsecureDomain()`. This method does the same thing as the `Security.allowDomain()` method, except that, when called from a SWF file served by a secure HTTPS connection, it additionally permits access to the calling SWF file by other SWF files that are served from an insecure protocol, such as HTTP. However, it is not a good security practice to allow scripting between files from a secure protocol (HTTPS) and those from insecure protocols (such as HTTP); doing so can open secure content to snooping and spoofing attacks. Here is how such attacks can work: since the `Security.allowInsecureDomain()` method allows access to your secure HTTPS data by SWF files served over HTTP connections, an attacker interposed between your HTTP server and your users could replace your HTTP SWF file with one of their own, which can then access your HTTPS data.

Another important security-related method is the `Security.loadPolicyFile()` method, which causes Flash Player to check for a cross-domain policy file at a nonstandard location. For more information, see "Website controls (cross-domain policy files)" on page 456.

Security sandboxes

Client computers can obtain individual SWF files from a number of sources, such as from external websites or from a local file system. Flash Player individually assigns SWF files and other resources, such as shared objects, bitmaps, sounds, videos, and data files, to security sandboxes based on their origin when they are loaded into Flash Player. The following sections describe the rules, enforced by Flash Player, that govern what a SWF file within a given sandbox can access.

For more information on security sandboxes, see the *Flash Player 9 Security* white paper.

Remote sandboxes

Flash Player classifies assets (including SWF files) from the Internet in separate sandboxes that correspond to their website origin domains. By default, these files are authorized to access any resources from their own server. Remote SWF files can be allowed to access additional data from other domains by explicit website and author permissions, such as cross-domain policy files and the `Security.allowDomain()` method. For details, see "Website controls (cross-domain policy files)" on page 456 and "Author (developer) controls" on page 460.

Remote SWF files cannot load any local files or resources.

For more information, see the *Flash Player 9 Security* white paper.

Local sandboxes

Local file describes any file that is referenced by using the `file:` protocol or a Universal Naming Convention (UNC) path. Local SWF files are placed into one of three local sandboxes:

- The local-with-filesystem sandbox—For security purposes, Flash Player places all local SWF files and assets in the local-with-file-system sandbox, by default. From this sandbox, SWF files can read local files (by using the URLLoader class, for example), but they cannot communicate with the network in any way. This assures the user that local data cannot be leaked out to the network or otherwise inappropriately shared.
- The local-with-networking sandbox—When compiling a SWF file, you can specify that it has network access when run as a local file (see "Setting the sandbox type of local SWF files" on page 463).These files are placed in the local-with-networking sandbox. SWF files that are assigned to the local-with-networking sandbox forfeit their local file access. In return, the SWF files are allowed to access data from the network. However, a local-with-networking SWF file is still not allowed to read any network-derived data unless permissions are present for that action, through a cross-domain policy file or a call to the `Security.allowDomain()` method. In order to grant such permission, a cross-domain policy file must grant permission to *all* domains by using `<allow-access-from domain="*"/>` or by using `Security.allowDomain("*")`. For more information, see "Website controls (cross-domain policy files)" on page 456 and "Author (developer) controls" on page 460.
- The local-trusted sandbox—Local SWF files that are registered as trusted (by users or by installer programs) are placed in the local-trusted sandbox. System administrators and users also have the ability to reassign (move) a local SWF file to or from the local-trusted sandbox based on security considerations (see "Administrative user controls" on page 452 and "User controls" on page 454). SWF files that are assigned to the local-trusted sandbox can interact with any other SWF files and can load data from anywhere (remote or local).

Communication between the local-with-networking and local-with-filesystem sandboxes, as well as communication between the local-with-filesystem and remote sandboxes, is strictly forbidden. Permission to allow such communication cannot be granted by a Flash application or by a user or administrator.

Scripting in either direction between local HTML files and local SWF files—for example, using the ExternalInterface class—requires that both the HTML file and SWF file involved be in the local-trusted sandbox. This is because the local security models for browsers differ from the Flash Player local security model.

SWF files in the local-with-networking sandbox cannot load SWF files in the local-with-filesystem sandbox. SWF files in the local-with-filesystem sandbox cannot load SWF files in the local-with-networking sandbox.

Setting the sandbox type of local SWF files

You can configure a SWF file for the local-with-filesystem sandbox or the local-with-networking sandbox by setting the use-network flag in the Flex compiler. For more information, see "About the application compiler options" on page 196 in *Building and Deploying Flex Applications*.

An end user or the administrator of a computer can specify that a local SWF file is trusted, allowing it to load data from all domains, both local and network. This is specified in the Global Flash Player Trust and User Flash Player Trust directories. For more information, see "Administrative user controls" on page 452 and "User controls" on page 454.

For more information on local sandboxes, see "Local sandboxes" on page 462.

The Security.sandboxType property

An author of a SWF file can use the read-only static `Security.sandboxType` property to determine the type of sandbox to which Flash Player has assigned the SWF file. The Security class includes constants that represent possible values of the `Security.sandboxType` property, as follows:

- `Security.REMOTE`—The SWF file is from an Internet URL, and operates under domain-based sandbox rules.
- `Security.LOCAL_WITH_FILE`—The SWF file is a local file, but it has not been trusted by the user and was not published with a networking designation. The SWF file can read from local data sources but cannot communicate with the Internet.
- `Security.LOCAL_WITH_NETWORK`—The SWF file is a local file and has not been trusted by the user, but it was published with a networking designation. The SWF can communicate with the Internet but cannot read from local data sources.
- `Security.LOCAL_TRUSTED`—The SWF file is a local file and has been trusted by the user, using either the Settings Manager or a Flash Player trust configuration file. The SWF file can both read from local data sources and communicate with the Internet.

Restricting networking APIs

You can control a SWF file's access to network functionality by setting the `allowNetworking` parameter in the `<object>` and `<embed>` tags in the HTML page that contains the SWF content.

Possible values of `allowNetworking` are:

- `"all"` (the default)—All networking APIs are permitted in the SWF.
- `"internal"`—The SWF file may not call browser navigation or browser interaction APIs, listed later in this section, but it may call any other networking APIs.
- `"none"`—The SWF file may not call browser navigation or browser interaction APIs, listed later in this section, and it cannot use any SWF-to-SWF communication APIs, also listed later.

Calling a prevented API throws a SecurityError exception.

To set the `allowNetworking` parameter, in the `<object>` and <embed> tags in the HTML page that contains a reference the SWF file, add the `allowNetworking` parameter and set its value, as shown in the following example:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
 codebase="http://fpdownload.macromedia.com/pub/shockwave/cabs/flash/
  swflash.cab#version=9,0,18,0"
 width="600" height="400" id="test" align="middle">
<param name="allowNetworking" value="none" />
<param name="movie" value="test.swf" />
<param name="bgcolor" value="#333333" />
<embed src="test.swf" allowNetworking="none" bgcolor="#333333"
  width="600" height="400"
  name="test" align="middle" type="application/x-shockwave-flash"
  pluginspage="http://www.macromedia.com/go/getflashplayer" />
</object>
```

An HTML page may also use a script to generate SWF-embedding tags. You need to alter the script so that it inserts the proper `allowNetworking` settings. HTML pages generated by Flash and Flex Builder use the `AC_FL_RunContent()` function to embed references to SWF files, and you need to add the `allowNetworking` parameter settings to the script, as in the following:

```
AC_FL_RunContent( ... "allowNetworking", "none", ...)
```

The following APIs are prevented when `allowNetworking` is set to `"internal"`:

- `navigateToURL()`
- `fscommand()`
- `ExternalInterface.call()`

An addition to those APIs on the previous list, the following APIs are also prevented when `allowNetworking` is set to `"none"`:

- `sendToURL()`
- `FileReference.download()`
- `FileReference.upload()`
- `Loader.load()`
- `LocalConnection.connect()`
- `LocalConnection.send()`
- `NetConnection.connect()`
- `NetStream.play()`
- `Security.loadPolicyFile()`
- `SharedObject.getLocal()`
- `SharedObject.getRemote()`
- `Socket.connect()`
- `Sound.load()`
- `URLLoader.load()`
- `URLStream.load()`
- `XMLSocket.connect()`

Even if the selected `allowNetworking` setting permits a SWF file to use a networking API, there may be other restrictions based on security sandbox limitations, as described in this chapter.

When `allowNetworking` is set to `"none"`, you cannot reference external media in an `<img>` tag in the `htmlText` property of a TextField object (a SecurityError exception is thrown).

Full-screen mode security

Flash Player 9.0.27.0 and later support full-screen mode, in which Flash content can fill the entire screen. To enter full-screen mode, the `displayState` property of the Stage is set to the `StageDisplayState.FULL_SCREEN` constant. For more information, see "Working with full-screen mode" on page 173.

For SWF files running in a browser, there are some security considerations.

To enable full-screen mode, in the `<object>` and `<embed>` tags in the HTML page that contains a reference to the SWF file, add the `allowFullScreen` parameter, with its value set to `"true"` (the default value is `"false"`), as shown in the following example:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
 codebase="http://fpdownload.macromedia.com/pub/shockwave/cabs/flash/
  swflash.cab#version=9,0,18,0"
 width="600" height="400" id="test" align="middle">
<param name="allowFullScreen" value="true" />
<param name="movie" value="test.swf" />
<param name="bgcolor" value="#333333" />
<embed src="test.swf" allowFullScreen="true" bgcolor="#333333"
  width="600" height="400"
  name="test" align="middle" type="application/x-shockwave-flash"
  pluginspage="http://www.macromedia.com/go/getflashplayer" />
</object>
```

An HTML page may also use a script to generate SWF-embedding tags. You must alter the script so that it inserts the proper `allowFullScreen` settings. HTML pages generated by Flash and Flex Builder use the `AC_FL_RunContent()` function to embed references to SWF files, and you need to add the `allowFullScreen` parameter settings, as in the following:

```
AC_FL_RunContent( ... "allowFullScreen", "true", ...)
```

The ActionScript that initiates full-screen mode can be called only in response to a mouse event or keyboard event. If it is called in other situations, Flash Player throws an exception.

Users cannot enter text in text input fields while in full-screen mode. All keyboard input and keyboard-related ActionScript is disabled while in full-screen mode, with the exception of the keyboard shortcuts (such as pressing the Esc key) that return the application to normal mode.

A message appears when the content enters full-screen mode, instructing the user how to exit and return to normal mode. The message appears for a few seconds and then fades out.

Calling the `displayState` property of a Stage object throws an exception for any caller that is not in the same security sandbox as the Stage owner (the main SWF file). For more information, see "Stage security" on page 472.

Administrators can disable full-screen mode for SWF files running in browsers by setting `FullScreenDisable = 1` in the mms.cfg file. For details, see "Administrative user controls" on page 452.

In a browser, a SWF file must be contained in an HTML page to allow full-screen mode.

Full-screen mode is always permitted in the stand-alone player or in a projector file.

Loading content

A SWF file can load the following types of content:

- SWF files
- Images
- Sound
- Video

Loading SWF files and images

You use the Loader class to load SWF files and images (JPG, GIF, or PNG files). Any SWF file, other than one in the local-with-filesystem sandbox, can load SWF files and images from any network domain. Only SWF files in local sandboxes can load SWF files and images from the local file system. However, files in the local-with-networking sandbox can only load local SWF files that are in the local-trusted or local-with-networking sandbox. SWF files in the local-with-networking sandbox load local content other than SWF files (such as images), however they cannot access data in the loaded content.

When loading a SWF file from an nontrusted source (such as a domain other than that of the Loader object's root SWF file), you may want to define a mask for the Loader object, to prevent the loaded content (which is a child of the Loader object) from drawing to portions of the Stage outside of that mask, as in the following code:

```
import flash.display.*;
import flash.net.URLRequest;
var rect:Shape = new Shape();
rect.graphics.beginFill(0xFFFFFF);
rect.graphics.drawRect(0, 0, 100, 100);
addChild(rect);
var ldr:Loader = new Loader();
ldr.mask = rect;
var url:String = "http://www.unknown.example.com/content.swf";
var urlReq:URLRequest = new URLRequest(url);
ldr.load(urlReq);
addChild(ldr);
```

When you call the `load()` method of the Loader object, you can specify a `context` parameter, which is a LoaderContext object. The LoaderContext class includes three properties that let you define the context of how the loaded content can be used:

- `checkPolicyFile`—Use this property only when loading an image file (not a SWF file). Specify this for an image file from a domain other than that of the file containing the Loader object. If you set this property to `true`, the Loader checks the origin server for a cross-domain policy file (see "Website controls (cross-domain policy files)" on page 456). If the server grants permission to the Loader domain, ActionScript from SWF files in the Loader domain can access data in the loaded image. In other words, you can use the `Loader.content` property to obtain a reference to the Bitmap object that represents the loaded image, or the `BitmapData.draw()` method to access pixels from the loaded image.
- `securityDomain`—Use this property only when loading a SWF file (not an image). Specify this for a SWF file from a domain other than that of the file containing the Loader object. Only two values are currently supported for the `securityDomain` property: `null` (the default) and `SecurityDomain.currentDomain`. If you specify `SecurityDomain.currentDomain`, this requests that the loaded SWF file be *imported* to the sandbox of the loading SWF file, meaning that it operates as though it had been loaded from the loading SWF file's own server. This is only permitted if a cross-domain policy file is found on the loaded SWF file's server, allowing access by the loading SWF file's domain. If the required policy file is found, the loader and loadee can freely script each other once the load begins, since they are in the same sandbox. Note that sandbox importing can mostly be replaced by performing an ordinary load and then having the loaded SWF file call the `Security.allowDomain()` method. This latter method may be easier to use, since the loaded SWF file will then be in its own natural sandbox, and thus able to access resources on its own actual server.
- `applicationDomain`—Use this property only when loading a SWF file written in ActionScript 3.0 (not an image or a SWF file written in ActionScript 1.0 or 2.0). When loading the file, you can specify that the file be placed into a particular application domain, rather than the default of being placed in a new application domain that is a child of the loading SWF file's application domain. Note that application domains are subunits of security domains, and thus you can specify a target application domain only if the SWF file that you are loading is from your own security domain, either because it is from your own server, or because you have successfully imported it into your security domain using the `securityDomain` property. If you specify an application domain but the loaded SWF file is part of a different security domain, the domain you specify in `applicationDomain` is ignored. For more information, see "ApplicationDomain class" on page 436.

For details, see "The LoaderContext class" on page 181.

An important property of a Loader object is the `contentLoaderInfo` property, which is a LoaderInfo object. Unlike most other objects, a LoaderInfo object is shared between the loading SWF file and the loaded content, and it is always accessible to both parties. When the loaded content is a SWF file, it can access the LoaderInfo object through the `DisplayObject.loaderInfo` property. LoaderInfo objects include information such as load progress, the URLs of loader and loadee, the trust relationship between loader and loadee, and other information. For more information, see "The LoaderInfo class" on page 180.

Loading sound and videos

All SWF files, other than those in the local-with-filesystem sandbox, are allowed to load sound and video from network origins, using the `Sound.load()`, `NetConnection.connect()`, and `NetStream.play()` methods.

Only local SWF files can load media from the local file system. Only SWF files in the local-with-filesystem sandbox or the local-trusted sandbox can access data in these loaded files.

There are other restrictions on accessing data from loaded media. For details, see "Accessing loaded media as data" on page 474.

Loading SWF files and images using the <img> tag in a text field

You can load SWF files and bitmaps into a text field by using the `<img>` tag, as in the following code:

```
<img src = 'filename.jpg' id = 'instanceName' >
```

You can access content loaded this way by using the `getImageReference()` method of the TextField instance, as in the following code:

```
var loadedObject:DisplayObject =
  myTextField.getImageReference('instanceName');
```

Note, however, that SWF files and images loaded in this way are put in the sandbox that corresponds to their origin.

When you load an image file using an `<img>` tag in a text field, access to the data in the image may be permitted by a cross-domain policy file. You can check for a policy file by adding a `checkPolicyFile` attribute to the `<img>` tag, as in the following code:

```
<img src = 'filename.jpg' checkPolicyFile = 'true' id = 'instanceName' >
```

When you load a SWF using an `<img>` tag in a text field, you can permit access to that SWF file's data through a call to the `Security.allowDomain()` method.

When you use an `<img>` tag in a text field to load an external file (as opposed to using a Bitmap class embedded within your SWF), a Loader object is automatically created as a child of the TextField object, and the external file is loaded into that Loader just as if you had used a Loader object in ActionScript to load the file. In this case, the `getImageReference()` method returns the Loader that was automatically created. No security check is needed to access this Loader object because it is in the same security sandbox as the calling code.

However, when you refer to the `content` property of the Loader object to access the loaded media, security rules apply. If the content is an image, you need to implement a cross-domain policy file, and if the content is a SWF file, you need to have the code in the SWF file call the `allowDomain()` method.

Content delivered using RTMP servers

Flash Media Server uses the Real-Time Media Protocol (RTMP) to serve data, audio, and video. A SWF file loads this media by using the `connect()` method of the NetConnection class, passing an RTMP URL as the parameter. Flash Media Server can restrict connections and prevent content from downloading, based on the domain of the requesting file. For details, see the Flash Media Server documentation.

For media loaded from RTMP sources, you cannot use the `BitmapData.draw()` and `SoundMixer.computeSpectrum()` methods to extract run-time graphics and sound data.

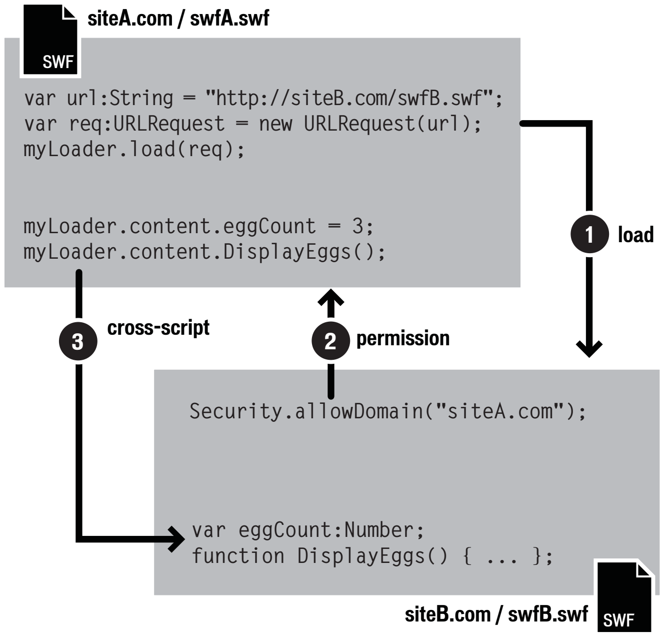
Cross-scripting

If two SWF files written with ActionScript 3.0 are served from the same domain—for example, the URL for one SWF file is http://www.example.com/swfA.swf and the URL for the other is http://www.example.com/swfB.swf—then one SWF file can examine and modify variables, objects, properties, methods, and so on in the other, and vice versa. This is called *cross-scripting*.

Cross-scripting is not supported between AVM1 SWF files and AVM2 SWF files. An AVM1 SWF file is one created by using ActionScript 1.0 or ActionScript 2.0. (AVM1 and AVM2 refer to the ActionScript Virtual Machine.) You can, however, use the LocalConnection class to send data between AVM1 and AVM2.

If two SWF files written with ActionScript 3.0 are served from different domains—for example, http://siteA.com/swfA.swf and http://siteB.com/swfB.swf—then, by default, Flash Player does not allow swfA.swf to script swfB.swf, nor swfB.swf to script swfA.swf. A SWF file gives permission to SWF files from other domains by calling `Security.allowDomain()`. By calling `Security.allowDomain("siteA.com")`, swfB.swf gives SWF files from siteA.com permission to script it.

In any cross-domain situation, it is important to be clear about the two parties involved. For the purposes of this discussion, the side that is performing the cross-scripting is called the *accessing party* (usually the accessing SWF), and the other side is called the *party being accessed* (usually the SWF being accessed). When siteA.swf scripts siteB.swf, siteA.swf is the accessing party, and siteB.swf is the party being accessed, as the following illustration shows:



```
var url:String = "http://siteB.com/swfB.swf";
var req:URLRequest = new URLRequest(url);
myLoader.load(req);

myLoader.content.eggCount = 3;
myLoader.content.DisplayEggs();
```

```
Security.allowDomain("siteA.com");

var eggCount:Number;
function DisplayEggs() { ... };
```

Cross-domain permissions that are established with the `Security.allowDomain()` method are asymmetrical. In the previous example, siteA.swf can script siteB.swf, but siteB.swf cannot script siteA.swf, because siteA.swf has not called the `Security.allowDomain()` method to give SWF files at siteB.com permission to script it. You can set up symmetrical permissions by having both SWF files call the `Security.allowDomain()` method.

In addition to protecting SWF files from cross-domain scripting originated by other SWF files, Flash Player protects SWF files from cross-domain scripting originated by HTML files. HTML-to-SWF scripting can occur with callbacks established through the `ExternalInterface.addCallback()` method. When HTML-to-SWF scripting crosses domains, the SWF file being accessed must call the `Security.allowDomain()` method, just as when the accessing party is a SWF file, or the operation will fail. For more information, see "Author (developer) controls" on page 460.

Also, Flash Player provides security controls for SWF-to-HTML scripting. For more information, see "Controlling access to scripts in a host web page" on page 481.

Stage security

Some properties and methods of the Stage object are available to any sprite or movie clip on the display list.

However, the Stage object is said to have an owner: the first SWF file loaded. By default, the following properties and methods of the Stage object are available only to SWF files in the same security sandbox as the Stage owner:

Properties		Methods
`align`	`showDefaultContextMenu`	`addChild()`
`displayState`	`stageFocusRect`	`addChildAt()`
`frameRate`	`stageHeight`	`addEventListener()`
`height`	`stageWidth`	`dispatchEvent()`
`mouseChildren`	`tabChildren`	`hasEventListener()`
`numChildren`	`textSnapshot`	`setChildIndex()`
`quality`	`width`	`willTrigger()`
`scaleMode`		

In order for a SWF file in a sandbox other than that of the Stage owner to access these properties and methods, the Stage owner SWF file must call the `Security.allowDomain()` method to permit the domain of the external sandbox. For more information, see "Author (developer) controls" on page 460.

The `frameRate` property is a special case—any SWF file can read the `frameRate` property. However, only those in the Stage owner's security sandbox (or those granted permission by a call to the `Security.allowDomain()` method) can change the property.

There are also restrictions on the `removeChildAt()` and `swapChildrenAt()` methods of the Stage object, but these are different from the other restrictions. Rather than needing to be in the same domain as the Stage owner, to call these methods code must be in the same domain as the owner of the affected child object(s), or the child object(s) can call the `Security.allowDomain()` method.

Traversing the display list

The ability of one SWF file to access display objects loaded from other sandboxes is restricted. In order for a SWF file to access a display object created by another SWF file in a different sandbox, the SWF file being accessed must call the `Security.allowDomain()` method to permit access by the domain of the accessing SWF file. For more information, see "Author (developer) controls" on page 460.

To access a Bitmap object that was loaded by a Loader object, a cross-domain policy file must exist on the origin server of the image file, and that cross-domain policy file must grant permission to the domain of the SWF file trying to access the Bitmap object (see "Website controls (cross-domain policy files)" on page 456).

The LoaderInfo object that corresponds to a loaded file (and to the Loader object) includes the following three properties, which define the relationship between the loaded object and the Loader object: `childAllowsParent`, `parentAllowsChild`, and `sameDomain`.

Event security

Events related to the display list have security access limitations, based on the sandbox of the display object that is dispatching the event. An event in the display list has bubbling and capture phases (described in Chapter 13, "Handling Events," on page 345). During the bubbling and capture phases, an event migrates from the source display object through parent display objects in the display list. If a parent object is in a different security sandbox than the source display object, the capture and bubble phase stops below that parent object, unless there is mutual trust between the owner of the parent object and the owner of the source object. This mutual trust can be achieved by the following:

1. The SWF file that owns the parent object must call the `Security.allowDomain()` method to trust the domain of the SWF file that owns the source object.
2. The SWF file that owns the source object must call the `Security.allowDomain()` method to trust the domain of the SWF file that owns the parent object.

The LoaderInfo object that corresponds to a loaded file (and to the Loader object) includes the following two properties, which define the relationship between the loaded object and the Loader object: `childAllowsParent` and `parentAllowsChild`.

For events that are dispatched from objects other than display objects, there are no security checks or security-related implications.

Accessing loaded media as data

You access loaded data using methods such as `BitmapData.draw()` and `SoundMixer.computeSpectrum()`. By default, a SWF file from one security sandbox cannot obtain pixel data or audio data from graphic or audio objects rendered or played by loaded media in another sandbox. However, you can use the following methods to grant this permission:

- In a loaded SWF file, call the `Security.allowDomain()` method to grant data access to SWF files in other domains.
- For a loaded image, sound, or video, add a cross-domain policy file on the server of the loaded file. This policy file must grant access to the domain of the SWF file that is attempting to call the `BitmapData.draw()` or `SoundMixer.computeSpectrum()` methods to extract data from the file.

The following sections provide details on accessing bitmap, sound, and video data.

Accessing bitmap data

The `draw()` method of a BitmapData object lets you draw the currently displayed pixels of any display object to the BitmapData object. This could include the pixels of a MovieClip object, a Bitmap object, or any display object. The following conditions must be met for the `draw()` method to draw pixels to the BitmapData object:

- In the case of a source object other than a loaded bitmap, the source object and (in the case of a Sprite or MovieClip object) all of its child objects must come from the same domain as the object calling the `draw()` method, or they must be in a SWF file that is accessible to the caller by having called the `Security.allowDomain()` method.
- In the case of a Loaded bitmap source object, the source object must come from the same domain as the object calling the `draw()` method, or its source server must include a cross-domain policy file that grants permission to the calling domain.

If these conditions are not met, a SecurityError exception is thrown.

When you load the image using the `load()` method of the Loader class, you can specify a `context` parameter, which is a LoaderContext object. If you set the `checkPolicyFile` property of the LoaderContext object to `true`, Flash Player checks for a cross-domain policy file on the server from which the image is loaded. If there is a cross-domain policy file, and the file permits the domain of the loading SWF file, the file is allowed to access data in the Bitmap object; otherwise, access is denied.

You can also specify a `checkPolicyFile` property in an image loaded via an `<img>` tag in a text field. For details, see "Loading SWF files and images using the <img> tag in a text field" on page 469.

Accessing sound data

The following sound-related ActionScript 3.0 APIs have security restrictions:

- The `SoundMixer.computeSpectrum()` method—Always permitted for SWF files that are in the same security sandbox as the sound file. For files in other sandboxes, there are security checks.
- The `SoundMixer.stopAll()` method—Always permitted for SWF files that are in the same security sandbox as the sound file. For files in other sandboxes, there are security checks.
- The `id3` property of the Sound class—Always permitted for SWF files that are in the same security sandbox as the sound file. For files in other sandboxes, there are security checks.

Every sound has two kinds of sandboxes associated with it—a content sandbox and an owner sandbox:

- The origin domain for the sound determines the content sandbox, and this determines whether data can be extracted from the sound via the `id3` property of the sound and the `SoundMixer.computeSpectrum()` method.
- The object that started the sound playing determines the owner sandbox, and this determines whether the sound can be stopped using the `SoundMixer.stopAll()` method.

When you load the sound using the `load()` method of the Sound class, you can specify a `context` parameter, which is a SoundLoaderContext object. If you set the `checkPolicyFile` property of the SoundLoaderContext object to `true`, Flash Player checks for a cross-domain policy file on the server from which the sound is loaded. If there is a cross-domain policy file, and the file permits the domain of the loading SWF file, the file is allowed to access the `id` property of the Sound object; otherwise, it will not. Also, setting the `checkPolicyFile` property can enable the `SoundMixer.computeSpectrum()` method for loaded sounds.

You can use the `SoundMixer.areSoundsInaccessible()` method to find out whether a call to the `SoundMixer.stopAll()` method would not stop all sounds because the sandbox of one or more sound owners is inaccessible to the caller.

Calling the `SoundMixer.stopAll()` method stops those sounds whose owner sandbox is the same as that of the caller of `stopAll()`. It also stops those sounds whose playback was started by SWF files that have called the `Security.allowDomain()` method to permit access by the domain of the SWF file calling the `stopAll()` method. Any other sounds are not stopped, and the presence of such sounds can be revealed by calling the `SoundMixer.areSoundsInaccessible()` method.

Calling the `computeSpectrum()` method requires that every sound that is playing be either from the same sandbox as the object calling the method or from a source that has granted permission to the caller's sandbox; otherwise, a SecurityError exception is thrown. For sounds that were loaded from embedded sounds in a library in a SWF file, permission is granted with a call to the `Security.allowDomain()` method in the loaded SWF file. For sounds loaded from sources other than SWF files (originating from loaded MP3 files or from Flash video), a cross-domain policy file on the source server grants access to data in loaded media. You cannot use the `computeSpectrum()` method if a sound is loaded from RTMP streams.

For more information, see "Author (developer) controls" on page 460 and "Website controls (cross-domain policy files)" on page 456.

Accessing video data

You can use the `BitmapData.draw()` method to capture the pixel data of the current frame of a video.

There are two different kinds of video:

- RTMP video
- Progressive video, which is loaded from an FLV file without an RTMP server

You cannot use the `BitmapData.draw()` method to access RTMP video.

When you call the `BitmapData.draw()` method with progressive video as the `source` parameter, the caller of `BitmapData.draw()` must either be from the same sandbox as the FLV file, or the server of the FLV file must have a policy file that grants permission to the domain of the calling SWF file. You can request that the policy file be downloaded by setting the `checkPolicyFile` property of the NetStream object to `true`.

Loading data

SWF files can load data from servers into ActionScript, and send data from ActionScript to servers. Loading data is a different kind of operation from loading media, because the loaded information appears directly in ActionScript, rather than being displayed as media. Generally, SWF files may load data from their own domains. However, they usually require cross-domain policy files in order to load data from other domains.

Using URLLoader and URLStream

You can load data, such as an XML file or a text file. The `load()` methods of the URLLoader and URLStream classes are governed by cross-domain policy file permissions.

If you use the `load()` method to load content from a domain other than that of the SWF file that is calling the method, Flash Player checks for a cross-domain policy file on the server of the loaded assets. If there is a cross-domain policy file, and it grants access to the domain of the loading SWF file, you can load the data.

Connecting to sockets

Cross-domain access to socket and XML socket connections is disabled by default. Also disabled by default is access to socket connections in the same domain as the SWF file on ports lower than 1024. You can permit access to these ports by serving a cross-domain policy file from any of the following locations:

- The same port as the main socket connection
- A different port
- An HTTP server on port 80 in the same domain as the socket server

If you serve the cross-domain policy file from same port as the main socket connection, or in a different port, you enumerate permitted ports by using a `to-ports` attribute in the cross-domain policy file, as the following example shows:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<!-- Policy file for xmlsocket://socks.mysite.com -->
<cross-domain-policy>
   <allow-access-from domain="*" to-ports="507" />
   <allow-access-from domain="*.example.com" to-ports="507,516" />
   <allow-access-from domain="*.example.org" to-ports="516-523" />
   <allow-access-from domain="adobe.com" to-ports="507,516-523" />
   <allow-access-from domain="192.0.34.166" to-ports="*" />
</cross-domain-policy>
```

To retrieve a socket policy file from the same port as a main socket connection, simply call the `Socket.connect()` or `XMLSocket.connect()` method, and, if the specified domain is not the same as the domain of the calling SWF file, Flash Player automatically attempts to retrieve a policy file from the same port as the main connection you are attempting. To retrieve a socket policy file from a different port on the same server as your main connection, call the `Security.loadPolicyFile()` method with the special `"xmlsocket"` syntax, as in the following:

```
Security.loadPolicyFile("xmlsocket://server.com:2525");
```

Call the `Security.loadPolicyFile()` method before calling the `Socket.connect()` or `XMLSocket.connect()` method. Flash Player then waits until it has fulfilled your policy file request before deciding whether to allow your main connection.

If you are implementing a socket server and you need to provide a socket policy file, decide whether you will provide the policy file using the same port that accepts main connections, or using a different port. In either case, your server must wait for the first transmission from your client before deciding whether to send a policy file or set up a main connection. When Flash Player requests a policy file, it always transmits the following string as soon as a connection is established:

```
<policy-file-request/>
```

Once the server receives this string, it can transmit the policy file. Do not expect to reuse the same connection for both a policy file request and a main connection; you should close the connection after transmitting the policy file. If you do not, Flash Player closes the policy file connection before reconnecting to set up the main connection.

For more information, see "Socket policy files" on page 458.

Sending data

Data sending occurs when ActionScript code from a SWF file sends data to a server or resource. Sending data is always permitted for network domain SWF files. A local SWF file can send data to network addresses only if it is in the local-trusted or local-with-networking sandbox. For more information, see "Local sandboxes" on page 462.

You can use the `flash.net.sendToURL()` function to send data to a URL. Other methods also send requests to URLs. These include loading methods, such as `Loader.load()` and `Sound.load()`, and data-loading methods, such as `URLLoader.load()` and `URLStream.load()`.

Uploading and downloading files

The `FileReference.upload()` method starts the upload of a file selected by a user to a remote server. You must call the `FileReference.browse()` or `FileReferenceList.browse()` method before calling the `FileReference.upload()` method.

Calling the `FileReference.download()` method opens a dialog box in which the user can download a file from a remote server.

NOTE

If your server requires user authentication, only SWF files running in a browser—that is, using the browser plug-in or ActiveX control—can provide a dialog box to prompt the user for a user name and password for authentication, and only for downloads. Flash Player does not allow uploads to a server that requires user authentication.

Uploads and downloads are not allowed if the calling SWF file is in the local-with-filesystem sandbox.

By default, a SWF file may not initiate an upload to, or a download from, a server other than its own. A SWF file may upload to, or download from, a different server if that server provides a cross-domain policy file that grants permission to the domain of the invoking SWF file.

Loading embedded content from SWF files imported into a security domain

When you load a SWF file, you can set the `context` parameter of the `load()` method of the Loader object that is used to load the file. This parameter takes a LoaderContext object. When you set the `securityDomain` property of this LoaderContext object to `Security.currentDomain`, Flash Player checks for a cross-domain policy file on the server of the loaded SWF file. If there is a cross-domain policy file, and it grants access to the domain of the loading SWF file, you can load the SWF file as imported media. In this way, the loading file can get access to objects in the library of the SWF file.

An alternative way for a SWF file to access classes in loaded SWF files from a different security sandbox is to have the loaded SWF file call the `Security.allowDomain()` method to grant access to the domain of the calling SWF file. You can add the call to the `Security.allowDomain()` method to the constructor method of the main class of the loaded SWF file, and then have the loading SWF file add an event listener to respond to the `init` event dispatched by the `contentLoaderInfo` property of the Loader object. When this event is dispatched, the loaded SWF file has called the `Security.allowDomain()` method in the constructor method, and classes in the loaded SWF file are available to the loading SWF file. The loading SWF file can retrieve classes from the loaded SWF file by calling `Loader.contentLoaderInfo.applicationDomain.getDefinition()`.

Working with legacy content

In Flash Player 6, the domain that is used for certain Flash Player settings is based on the trailing portion of the domain of the SWF file. These settings include settings for camera and microphone permissions, storage quotas, and storage of persistent shared objects.

If the domain of a SWF file includes more than two segments, such as www.example.com, the first segment of the domain (www) is removed, and the remaining portion of the domain is used. So, in Flash Player 6, www.example.com and store.example.com both use example.com as the domain for these settings. Similarly, www.example.co.uk and store.example.co.uk both use example.co.uk as the domain for these settings. This can lead to problems in which SWF files from unrelated domains, such as example1.co.uk and example2.co.uk, have access to the same shared objects.

In Flash Player 7 and later, player settings are chosen by default according to a SWF file's exact domain. For example, a SWF file from www.example.com would use the player settings for www.example.com. A SWF file from store.example.com would use the separate player settings for store.example.com.

In a SWF file written using ActionScript 3.0, when `Security.exactSettings` is set to `true` (the default), Flash Player uses exact domains for player settings. When it is set to `false`, Flash Player uses the domain settings used in Flash Player 6. If you change `exactSettings` from its default value, you must do so before any events occur that require Flash Player to choose player settings—for example, using a camera or microphone, or retrieving a persistent shared object.

If you published a version 6 SWF file and created persistent shared objects from it, to retrieve those persistent shared objects from a SWF that uses ActionScript 3.0, you must set `Security.exactSettings` to `false` before calling `SharedObject.getLocal()`.

Setting LocalConnection permissions

The LocalConnection class lets you develop SWF files that can send instructions to each other. LocalConnection objects can communicate only among SWF files that are running on the same client computer, but they can be running in different applications—for example, a SWF file running in a browser and a SWF file running in a projector.

For every LocalConnection communication, there is a sender SWF file and a listener SWF file. By default, Flash Player allows LocalConnection communication between SWF files in the same domain. For SWF files in different sandboxes, the listener must allow the sender permission by using the `LocalConnection.allowDomain()` method. The string you pass as an argument to the `LocalConnection.allowDomain()` method can contain any of the following: exact domain names, IP addresses, and the * wildcard.

NOTE

The `allowDomain()` method has changed from the form it had in ActionScript 1.0 and 2.0. In those earlier versions, `allowDomain()` was a callback method that you implemented. In ActionScript 3.0, `allowDomain()` is a built-in method of the LocalConnection class that you call. With this change, `allowDomain()` works in much the same way as `Security.allowDomain()`.

A SWF file can use the `domain` property of the LocalConnection class to determine its domain.

Controlling access to scripts in a host web page

Outbound scripting is achieved through use of the following ActionScript 3.0 APIs:

- The `flash.system.fscommand()` function
- The `flash.net.navigateToURL()` function (when specifying a scripting statement, such as `navigateToURL("javascript: alert('Hello from Flash Player.')")`)
- The `flash.net.navigateToURL()` function (when the `window` parameter is set to `"_top"`, `"_self"`, or `"_parent"`)
- The `ExternalInterface.call()` method

For SWF files running locally, calls to these methods are sucessful only if the SWF file and the containing web page (if there is one) are in the local-trusted security sandbox. Calls to these methods fail if the content is in the local-with-networking or local-with-filesystem sandbox.

The `AllowScriptAccess` parameter in the HTML code that loads a SWF file controls the ability to perform outbound scripting from within a SWF file.

Set this parameter in the HTML code for the web page that hosts a SWF file. You set the parameter in the `PARAM` or `EMBED` tag.

The `AllowScriptAccess` parameter can have one of three possible values: `"always"`, `"sameDomain"`, or `"never"`:

- When `AllowScriptAccess` is `"sameDomain"`, outbound scripting is allowed only if the SWF file and the web page are in the same domain. This is the default for AVM2 content.
- When `AllowScriptAccess` is `"never"`, outbound scripting always fails.
- When `AllowScriptAccess` is `"always"`, outbound scripting always succeeds.

If the `AllowScriptAccess` parameter is not specified for a SWF file in an HTML page, it defaults to `"sameDomain"` for AVM2 content.

Here is an example of setting the `AllowScriptAccess` tag in an HTML page:

```
<object id='MyMovie.swf' classid='clsid:D27CDB6E-AE6D-11cf-96B8-
  444553540000' codebase='http://download.adobe.com/pub/shockwave/cabs/
  flash/swflash.cab#version=9,0,0,0' height='100%' width='100%'>
<param name='AllowScriptAccess' value='never'/>
<param name='src' value=''MyMovie.swf'/>
<embed name='MyMovie.swf' pluginspage='http://www.adobe.com/go/
  getflashplayer' src='MyMovie.swf' height='100%' width='100%'
  AllowScriptAccess='never'/>
</object>
```

The `AllowScriptAccess` parameter can prevent a SWF file hosted from one domain from accessing a script in an HTML page that comes from another domain. Using `AllowScriptAccess="never"` for all SWF files hosted from another domain can ensure the security of scripts located in an HTML page.

For more information, see the following entries in the *ActionScript 3.0 Language Reference:*

- The `flash.system.fscommand()` function
- The `flash.net.navigateToURL()` function
- The `call()` method of the ExternalInterface class

Shared objects

Flash Player provides the ability to use *shared objects*, which are ActionScript objects that persist outside of a SWF file, either locally on a user's file system or remotely on an RTMP server. Shared objects, like other media in Flash Player, are partitioned into security sandboxes. However, the sandbox model for shared objects is somewhat different, because shared objects are not resources that can ever be accessed across domain boundaries. Instead, shared objects are always retrieved from a shared object store that is particular to the domain of each SWF file that calls methods of the SharedObject class. Usually a shared object store is even more particular than a SWF file's domain: by default, each SWF file uses a shared object store particular to its entire origin URL.

A SWF file can use the `localPath` parameter of the `SharedObject.getLocal()` and `SharedObject.getRemote()` methods to use a shared object store associated with only a part of its URL. In this way, the SWF file can permit sharing with other SWF files from other URLs. Even if you pass `'/'` as the `localPath` parameter, this still specifies a shared object store particular to its own domain.

Users can restrict shared object access by using the Flash Player Settings dialog box or the Settings Manager. By default, shared objects can be created up to a maximum of 100 KB of data per domain. Administrative users and users can also place restrictions on the ability to write to the file system. For more information, see "Administrative user controls" on page 452 and "User controls" on page 454.

You can specify that a shared object is secure, by specifying `true` for the `secure` parameter of the `SharedObject.getLocal()` method or the `SharedObject.getRemote()` method. Note the following about the `secure` parameter:

- If this parameter is set to `true`, Flash Player creates a new secure shared object or gets a reference to an existing secure shared object. This secure shared object can be read from or written to only by SWF files delivered over HTTPS that call `SharedObject.getLocal()` with the `secure` parameter set to `true`.
- If this parameter is set to `false`, Flash Player creates a new shared object or gets a reference to an existing shared object that can be read from or written to by SWF files delivered over non-HTTPS connections.

If the calling SWF file is not from an HTTPS URL, specifying `true` for the `secure` parameter of the `SharedObject.getLocal()` method or the `SharedObject.getRemote()` method results in a SecurityError exception.

The choice of a shared object store is based on a SWF file's origin URL. This is true even in the two situations where a SWF file does not originate from a simple URL: import loading and dynamic loading. Import loading refers to the situation where you load a SWF file with the `LoaderContext.securityDomain` property set to `SecurityDomain.currentDomain`. In this situation, the loaded SWF file will have a pseudo-URL that begins with its loading SWF file's domain and then specifies its actual origin URL. Dynamic loading refers to the loading of a SWF file using the `Loader.loadBytes()` method. In this situation, the loaded SWF file will have a pseudo-URL that begins with its loading SWF file's full URL followed by an integer ID. In both the import loading and dynamic loading cases, a SWF file's pseudo-URL can be examined using the `LoaderInfo.url` property. The pseudo-URL is treated exactly like a real URL for the purposes of choosing a shared object store. You can specify a shared object `localPath` parameter that uses part or all of the pseudo-URL.

Users and administrators can elect to disable the use of *third-party shared objects*. This is the usage of shared objects by any SWF file that is executing in a web browser, when that SWF file's origin URL is from a different domain than the URL shown in the browser's address bar. Users and administrators may choose to disable third-party shared object usage for reasons of privacy, wishing to avoid cross-domain tracking. In order to avoid this restriction, you may wish to ensure that any SWF file using shared objects is loaded only within HTML page structures that ensure that the SWF file comes from the same domain as is shown in the browser's address bar. When you attempt to use shared objects from a third-party SWF file, and third-party shared object use is disabled, the `SharedObject.getLocal()` and `SharedObject.getRemote()` methods return `null`. For more information, see www.adobe.com/products/flashplayer/articles/thirdpartylso.

Camera, microphone, Clipboard, mouse, and keyboard access

When a SWF file attempts to access a user's camera or microphone using the `Camera.get()` or `Microphone.get()` methods, Flash Player displays a Privacy dialog box, in which the user can allow or deny access to their camera and microphone. The user and the administrative user can also disable camera access on a per-site or global basis, through controls in the mms.cfg file, the Settings UI, and the Settings Manager (see "Administrative user controls" on page 452 and "User controls" on page 454). With user restrictions, the `Camera.get()` and `Microphone.get()` methods each return a `null` value. You can use the `Capabilities.avHardwareDisable` property to determine whether the camera and microphone have been administratively prohibited (`true`) or allowed (`false`).

The `System.setClipboard()` method allows a SWF file to replace the contents of the Clipboard with a plain-text string of characters. This poses no security risk. To protect against the risk posed by passwords and other sensitive data being cut or copied to Clipboards, there is no corresponding “getClipboard” (read) method.

A Flash application can monitor only keyboard and mouse events that occur within its focus. A Flash application cannot detect keyboard or mouse events in another application.

CHAPTER 18

Printing

18

Adobe Flash Player 9 can communicate with an operating system's printing interface so that you can pass pages to the print spooler. Each page Flash Player sends to the spooler can contain content that is visible, dynamic, or offscreen to the user, including database values and dynamic text. Additionally, Flash Player sets the properties of the flash.printing.PrintJob class based on a user's printer settings, so that you can format pages appropriately.

This chapter details strategies for using the flash.printing.PrintJob class methods and properties to create a print job, read a user's print settings, and make adjustments to a print job based on feedback from Flash Player and the user's operating system.

Contents

What's new for the PrintJob class using ActionScript 3.0

The PrintJob class hasn't changed dramatically for the ActionScript 3.0 implementation of printing. However, a few critical differences are worth noting:

- The `PrintJob.addPage()` method now takes a Sprite and a Rectangle object for the first two parameters.
- ActionScript 3.0 does not support using the `delete` (as in `delete myPrintJob`) operator to remove an entire object. In ActionScript 2.0, you could use `delete` to remove an object or an object property. In ActionScript 3.0, the `delete` operator is now ECMAScript-compatible, meaning `delete` can be used only on a dynamic property of an object. The `PrintJob.start()` method resets an existing PrintJob object's properties, so you do not need to delete any existing PrintJob objects to reuse the same variable. If you do need to "clear" a PrintJob object's properties for any reason, use `null` (as in `myPrintJob = null`).
- The global ActionScript 2.0 functions `print()`, `printAsBitmap()`, `printAsBitmapNum()`, and `printNum()` are not supported in ActionScript 3.0. Use the PrintJob and PrintJobOptions classes instead.
- Because Flash Player now throws run-time exceptions for certain `PrintJob.addPage()` conditions, you can write code to catch and handle the exceptions as they occur.
- ActionScript 3.0 does not restrict a PrintJob object to a single frame (but script time-out limits do apply, as detailed in "Timing print job statements" on page 492).

Printing a page

The ActionScript to print a basic page through Flash Player requires four major statements in sequence:

- `new PrintJob()`–Creates a new print job instance of the name you specify.
- `PrintJob.start()`–Initiates the printing process for the operating system, invoking the print dialog box for the user, and populates the read-only properties of the print job.
- `PrintJob.addPage()`–Contains the details about the print job contents, including the Sprite object (and any children it contains), the size of the print area, and whether the printer should print the image as a vector or bitmap. You can use successive calls to `addPage()` to print multiple sprites over several pages.
- `PrintJob.send()`–Sends the pages to the operating system's printer.

So, for example, a very simple print job script may look like the following (including `package`, `import` and `class` statements for compiling):

```
package
{
  import flash.printing.PrintJob;
  import flash.display.Sprite;

  public class BasicPrintExample extends Sprite
  {
    var myPrintJob:PrintJob = new PrintJob();
    var mySprite:Sprite = new Sprite();

    public function BasicPrintExample()
    {
      myPrintJob.start();
      myPrintJob.addPage(mySprite);
      myPrintJob.send();
    }
  }
}
```

NOTE

This example is intended to show the basic elements of a print job script, and does not contain any error handling. To build a script that responds properly to a user canceling a print job, see "Working with exceptions and returns" on page 490.

Flash Player tasks and system printing

Because Flash Player dispatches pages to the operating system's printing interface, you should understand the scope of the tasks managed by Flash Player, and the tasks managed by an operating system's own printing interface. Flash Player can initiate a print job, read some of a printer's page settings, pass the content for a print job to the operating system, and verify if the user or system has cancelled a print job. Other processes, such as displaying printer specific dialog boxes, cancelling a spooled print job, or reporting on the printer's status, are all handled by the operating system. Flash Player is able to respond if there is a problem initiating or formatting a print job, but can report back only on certain properties or conditions from the operating system's printing interface. As a developer, your code should have the ability to respond to these properties or conditions.

Working with exceptions and returns

You should check to see if the `PrintJob.start()` method returns `true` before executing `addPage()` and `send()` calls, in case the user has cancelled the print job. A simple way to check whether these methods have been cancelled before continuing is to wrap them in an `if` statement, as follows:

```
if (myPrintJob.start())
{
  // addPage() and send() statements here
}
```

If `PrintJob.start()` is `true`, meaning the user has selected Print (or Flash Player has initiated a Print command), the `addPage()` and `send()` methods can be called.

Also, to help manage the printing process, Flash Player now throws exceptions for the `PrintJob.addPage()` method, so that you can catch errors and provide information and options to the user. If a `PrintJob.addPage()` method fails, you can also call another function or stop the current print job. You catch these exceptions by embedding `addPage()` calls within a `try..catch` statement, as in the following example. In the example, `[params]` is a placeholder for the parameters specifying the actual content you want to print.

```
if (myPrintJob.start())
{
  try
  {
    myPrintJob.addPage([params]);
  }
  catch (e:Error)
  {
    // Handle error,
  }
  myPrintJob.send();
}
```

Once the print job starts, you can add the content using `PrintJob.addPage()` and see if that generates an exception (for example, if the user has cancelled the print job). If it does, you can add logic to the `catch` statement to provide the user (or Flash Player) with information, options, or you can stop the current print job. If you add the page successfully, you can proceed to send the pages to the printer using `PrintJob.send()`.

If Flash Player encounters a problem sending the print job to the printer (for example, if the printer is offline), you can catch that exception, too, and provide the user (or Flash Player) with information or more options (such as displaying message text or providing an alert within the Flash animation). For example, you can assign new text to a text field in an `if..else` statement, as the following code shows:

```
if (myPrintJob.start())
{
  try
  {
    myPrintJob.addPage([params]);
  }
  catch (e:Error)
  {
    // Handle error.
  }
  myPrintJob.send();
}
else
{
  myAlert.text = "Print job canceled";
}
```

For a working example, see “Example: Scaling, cropping, and responding” on page 498.

Working with page properties

Once the user clicks OK in Print dialog box and `PrintJob.start()` returns `true`, you can access the properties defined by the printer's settings. These include the paper width, paper height (`pageHeight` and `pageWidth`), and content orientation on the paper. Because these are printer settings, not controlled by Flash Player, you cannot alter these settings; however, you can use them to align the content you send to the printer to match the current settings. For more information, see “Setting size, scale, and orientation” on page 493.

Setting vector or bitmap rendering

You can manually set the print job to spool each page as vector graphics or a bitmap image. In some cases, vector printing will produce a smaller spool file, and a better image than bitmap printing. However, if your content includes a bitmap image, and you want to preserve any alpha transparency or color effects, you should print the page as a bitmap image. Also, a non-PostScript printer automatically converts any vector graphics to bitmap images. You specify bitmap printing in the third parameter of `PrintJob.addPage()`, by passing a PrintJobOptions object with the `printAsBitmap` parameter set to `true`, as follows:

```
var options:PrintJobOptions = new PrintJobOptions();
options.printAsBitmap = true;
myPrintJob.addPage(mySprite, null, options);
```

If you don't specify a value for the third parameter, the print job will use the default, which is vector printing.

NOTE

If you don't want to specify a value for `printArea` (the second parameter) but want to specify a value for bitmap printing, use `null` for `printArea`.

Timing print job statements

ActionScript 3.0 does not restrict a PrintJob object to a single frame (as did previous versions of ActionScript). However, because the operating system displays print status information to the user once the user has clicked the OK button in the Print dialog box, you should call `PrintJob.addPage()` and `PrintJob.send()` as soon as possible to send pages to the spooler. A delay reaching the frame containing the `PrintJob.send()` call will delay the printing process.

Because ActionScript 2.0 does have a single-frame restriction, any print job statements in ActionScript 2.0 are restricted to 15 seconds before hitting the script time-out limit. In ActionScript 3.0, this script time-out limit still applies. Therefore, the time between each major statement in a print job sequence cannot exceed 15 seconds. In other words, the 15-second script time-out limit applies to the following intervals:

- Between `PrintJob.start()` and the first `PrintJob.addPage()`
- Between `PrintJob.addPage()` and the next `PrintJob.addPage()`
- Between the last `PrintJob.addPage()` and `PrintJob.send()`

If any of these intervals spans more than 15 seconds, the next call to `PrintJob.start()` on the PrintJob instance returns `false`, and the next `PrintJob.addPage()` on the PrintJob instance causes Flash Player to throw a run-time exception.

Setting size, scale, and orientation

The section "Printing a page" on page 488 details the steps for a basic print job, where the output directly reflects the printed equivalent of the screen size and position of the specified sprite. However, printers use different resolutions for printing, and can have settings that adversely affect the appearance of the printed sprite.

Flash Player can read an operating system's printing settings, but note that these properties are read-only: although you can respond to their values, you can't set them. So, for example, you can find out the printer's page size setting and adjust your content to fit the size. You can also determine a printer's margin settings and page orientation. To respond to the printer settings, you may need to specify a print area, adjust for the difference between a screen's resolution and a printer's point measurements, or transform your content to meet the size or orientation settings of the user's printer.

Using rectangles for the print area

The `PrintJob.addPage()` method allows you to specify the region of a sprite that you want printed. The second parameter, `printArea`, is in the form of a Rectangle object. You have three options for providing a value for this parameter:

- Create a Rectangle object with specific properties and then use that rectangle in the `addPage()` call, as in the following example:

  ```
  private var rect1:Rectangle = new Rectangle(0,0,400,200);
  myPrintJob.addPage(sheet, rect1);
  ```

- If you haven't already specified a Rectangle object, you can do it within the call itself, as in the following example:

  ```
  myPrintJob.addPage(sheet, new Rectangle(0, 0, 100, 100));
  ```

- If you plan to provide values for the third parameter in the `addPage()` call, but don't want to specify a rectangle, you can use `null` for the second parameter, as in the following;

  ```
  myPrintJob.addPage(sheet, null, options);
  ```

NOTE

If you plan to specify a rectangle for the printing dimensions, remember to import the `flash.display.Rectangle` class.

Comparing points and pixels

A rectangle's width and height are pixel values. A printer uses points as print units of measurement. Points are a fixed physical size (1/72 inch), but the size of a pixel on the screen depends on the resolution of the particular screen. The conversion rate between pixels and points depends on the printer settings and whether the sprite is scaled. An unscaled sprite that is 72 pixels wide will print out one inch wide, with one point equal to one pixel, independent of screen resolution.

You can use the following equivalencies to convert inches or centimeters to twips or points (a twip is 1/20 of a point):

- 1 point = 1/72 inch = 20 twips
- 1 inch = 72 points = 1440 twips
- 1 centimeter = 567 twips

If you omit the `printArea` parameter, or if it is passed incorrectly, the full area of the sprite is printed.

Scaling

If you want to scale a Sprite object before you print it, set the scale properties (see the flash.display.DisplayObject class in the *ActionScript 3.0 Language Reference*) before calling the `PrintJob.addPage()` method, and set them back to their original values after printing. The scale of a Sprite object has no relation to the `printArea` property. In other words, if you specify a print area that is 50 pixels by 50 pixels, 2500 pixels are printed. If you scale the Sprite object, the same 2500 pixels are printed, but the Sprite object is printed at the scaled size.

For an example, see “Example: Scaling, cropping, and responding” on page 498.

Printing for landscape or portrait orientation

Because Flash Player can detect the settings for orientation, you can build logic into your ActionScript to adjust the content size or rotation in response to the printer settings, as the following example illustrates:

```
if (myPrintJob.orientation == PrintJobOrientation.LANDSCAPE)
{
  mySprite.rotation = 90;
}
```

NOTE

If you plan to read the system setting for content orientation on the paper, remember to import the PrintJobOrientation class by using the following:

`import flash.printing.PrintJobOrientation;`

The PrintJobOrientation class provides constant values that define the content orientation on the page.

Responding to page height and width

Using a strategy that is similar to handling printer orientation settings, you can read the page height and width settings and respond to them by embedding some logic into an `if` statement. The following code shows an example:

```
if (mySprite.height > myPrintJob.pageHeight)
{
  mySprite.scaleY = .75;
}
```

In addition, a page's margin settings can be determined by comparing the page and paper dimensions, as the following example illustrates:

```
margin_height = (myPrintJob.paperHeight - myPrintJob.pageHeight) / 2;
margin_width = (myPrintJob.paperWidth - myPrintJob.pageWidth) / 2;
```

Example: Multiple-page printing

When printing more than one page of content, you can associate each page of content with a different sprite (in this case, `sheet1` and `sheet2`), and then use `PrintJob.addPage()` for each sprite. The following code illustrates this technique:

```
package
{
  import flash.display.MovieClip;
  import flash.printing.PrintJob;
  import flash.printing.PrintJobOrientation;
  import flash.display.Stage;
  import flash.display.Sprite;
  import flash.text.TextField;
import flash.geom.Rectangle;

public class PrintMultiplePages extends MovieClip
{
  private var sheet1:Sprite;
  private var sheet2:Sprite;

  public function PrintMultiplePages():void
  {
    init();
    printPages();
  }

  private function init():void
  {
    sheet1 = new Sprite();
    createSheet(sheet1, "Once upon a time...", {x:10, y:50, width:80,
height:130});
    sheet2 = new Sprite();
    createSheet(sheet2, "There was a great story to tell, and it ended
quickly.\n\nThe end.", null);
  }

  private function createSheet(sheet:Sprite, str:String,
imgValue:Object):void
  {
    sheet.graphics.beginFill(0xEEEEEE);
    sheet.graphics.lineStyle(1, 0x000000);
    sheet.graphics.drawRect(0, 0, 100, 200);
    sheet.graphics.endFill();

    var txt:TextField = new TextField();
    txt.height = 200;
    txt.width = 100;
    txt.wordWrap = true;
    txt.text = str;

    if (imgValue != null)
    {
      var img:Sprite = new Sprite();
      img.graphics.beginFill(0xFFFFFF);
      img.graphics.drawRect(imgValue.x, imgValue.y, imgValue.width,
imgValue.height);
      img.graphics.endFill();
      sheet.addChild(img);
    }
    sheet.addChild(txt);
  }

  private function printPages():void
```

```
    {
      var pj:PrintJob = new PrintJob();
      var pagesToPrint:uint = 0;
      if (pj.start())
      {
        if (pj.orientation == PrintJobOrientation.LANDSCAPE)
        {
          throw new Error("Page is not set to an orientation of
  portrait.");
        }

        sheet1.height = pj.pageHeight;
        sheet1.width = pj.pageWidth;
        sheet2.height = pj.pageHeight;
        sheet2.width = pj.pageWidth;

        try
        {
          pj.addPage(sheet1);
          pagesToPrint++;
        }
        catch (e:Error)
        {
        // Respond to error.
        }

        try
        {
          pj.addPage(sheet2);
          pagesToPrint++;
        }
        catch (e:Error)
        {
          // Respond to error.
        }

        if (pagesToPrint > 0)
        {
          pj.send();
        }
      }
    }
  }
}
```

Example: Scaling, cropping, and responding

In some cases, you may want adjust the size (or other properties) of a display object when printing it to accommodate differences between the way it appears on screen and the way it appears printed on paper. When you adjust the properties of a display object before printing (for example, by using the `scaleX` and `scaleY` properties), be aware that if the object scales larger than the defined rectangle for the print area, the object will be cropped. You will also probably want to reset the properties after the pages have been printed.

The following code scales the dimensions of the `txt` display object (but not the green box background), and the text field ends up being cropped by the dimensions of the specified rectangle. After printing, the text field is returned to its original size for display on screen. If the user cancels the print job from the operating system's Print dialog box, the content in the Flash Player changes to alert the user that the job has been canceled.

```
package
{
  import flash.printing.PrintJob;
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.display.Stage;
  import flash.geom.Rectangle;

  public class PrintScaleExample extends Sprite
  {
    private var bg:Sprite;
    private var txt:TextField;

    public function PrintScaleExample():void
    {
      init();
      draw();
      printPage();
    }

    private function printPage():void
    {
      var pj:PrintJob = new PrintJob();
      txt.scaleX = 3;
      txt.scaleY = 2;
      if (pj.start())
      {
        trace(">> pj.orientation: " + pj.orientation);
        trace(">> pj.pageWidth: " + pj.pageWidth);
        trace(">> pj.pageHeight: " + pj.pageHeight);
        trace(">> pj.paperWidth: " + pj.paperWidth);
```

```
          trace(">> pj.paperHeight: " + pj.paperHeight);

          try
          {
            pj.addPage(this, new Rectangle(0, 0, 100, 100));
          }
          catch (e:Error)
          {
            // Do nothing.
          }
          pj.send();
        }
        else
        {
          txt.text = "Print job canceled";
        }
        // Reset the txt scale properties.
        txt.scaleX = 1;
        txt.scaleY = 1;
      }

      private function init():void
      {
        bg = new Sprite();
        bg.graphics.beginFill(0x00FF00);
        bg.graphics.drawRect(0, 0, 100, 200);
        bg.graphics.endFill();

        txt = new TextField();
        txt.border = true;
        txt.text = "Hello World";
      }

      private function draw():void
      {
        addChild(bg);
        addChild(txt);
        txt.x = 50;
        txt.y = 50;
      }
    }
}
```

CHAPTER 19

Using the External API

19

The ActionScript 3.0 External API enables straightforward communication between ActionScript and the container application within which Adobe Flash Player 9 is running. There are several situations in which you may want to use the External API—for example, when you create interaction between a SWF document and JavaScript in an HTML page, or when building a desktop application that uses Flash Player to display a SWF file.

This chapter describes how to use the External API to interact with a container application, how to pass data between ActionScript and JavaScript in an HTML page, and how to establish communication and exchange data between ActionScript and a desktop application.

Contents

About the External API

The External API is the portion of ActionScript that provides a mechanism for communication between ActionScript and code running in an "external application" that is acting as a container for Flash Player (commonly a web browser or stand-alone projector application). In ActionScript 3.0, the functionality of the External API is provided by the ExternalInterface class. In Flash Player versions prior to Flash Player 8, the `fscommand()` action was used to carry out communication with the container application. The ExternalInterface class is a replacement for `fscommand()`, and its use is recommended for all communication between JavaScript and ActionScript.

NOTE

If you need to use the old `fscommand()` function–for example, to maintain compatibility with older applications or to interact with a third-party SWF container application or the stand-alone Flash Player–it is still available as a package-level function in the flash.system package.

The ExternalInterface class is a subsystem that lets you easily communicate from ActionScript and Flash Player to JavaScript in an HTML page, or to any desktop application that embeds Flash Player.

The ExternalInterface class is available only under the following conditions:

- In all supported versions of Internet Explorer for Windows (5.0 and later).
- In an embedded custom ActiveX container, such as a desktop application embedding the Flash Player ActiveX control.
- In any browser that supports the NPRuntime interface, which currently includes the following browsers:
  - Firefox 1.0 and later
  - Mozilla 1.7.5 and later
  - Netscape 8.0 and later
  - Safari 1.3 and later

In all other situations (such as running in a stand-alone player), the `ExternalInterface.available` property returns `false`.

From ActionScript, you can call a JavaScript function on the HTML page. The External API offers the following improved functionality compared with `fscommand()`:

- You can use any JavaScript function, not only the functions that you can use with the `fscommand()` function.
- You can pass any number of arguments, with any names; you aren't limited to passing a command and a single string argument. This gives the External API much more flexibility than `fscommand()`.
- You can pass various data types (such as Boolean, Number, and String); you are no longer limited to String parameters.
- You can now receive the value of a call, and that value returns immediately to ActionScript (as the return value of the call you make).

WARNING

If the name given to the Flash Player instance in an HTML page (the `<object>` tag's `id` attribute) includes a hyphen (`-`) or other characters that are defined as operators in JavaScript (such as `+`, `*`, `/`, `\`, `.`, and so on), ExternalInterface calls from ActionScript will not work when the container web page is viewed in Internet Explorer.
In addition, if the HTML tags that define the Flash Player instance (the `<object>` and `<embed>` tags) are nested in an HTML `<form>` tag, ExternalInterface calls from ActionScript will not work.

Using the ExternalInterface class

Communication between ActionScript and the container application can take one of two forms: either ActionScript can call code (such as a JavaScript function) defined in the container, or code in the container can call an ActionScript function that has been designated as being callable. In either case, information can be sent to the code being called, and results can be returned to the code making the call.

To facilitate this communication, the ExternalInterface class includes two static properties and two static methods. These properties and methods are used to obtain information about the external interface connection, to execute code in the container from ActionScript, and to make ActionScript functions available to be called by the container.

Getting information about the external container

The `ExternalInterface.available` property indicates whether the current Flash Player is in a container that offers an external interface. If the external interface is available, this property is `true`; otherwise, it is `false`. Before using any of the other functionality in the ExternalInterface class, you should always check to make sure that the current container supports external interface communication, as follows:

```
if (ExternalInterface.available)
{
  // Perform ExternalInterface method calls here.
}
```

NOTE: The `ExternalInterface.available` property reports whether the current container is a type that supports ExternalInterface connectivity. It will not tell you if JavaScript is enabled in the current browser.

The `ExternalInterface.objectID` property allows you to determine the unique identifier of the Flash Player instance (specifically, the `id` attribute of the `<object>` tag in Internet Explorer or the `name` attribute of the `<embed>` tag in browsers using the NPRuntime interface). This unique ID represents the current SWF document in the browser, and can be used to make reference to the SWF document—for example, when calling a JavaScript function in a container HTML page. When the Flash Player container is not a web browser, this property is `null`.

Calling external code from ActionScript

The `ExternalInterface.call()` method executes code in the container application. It requires at least one parameter, a string containing the name of the function to be called in the container application. Any additional parameters passed to the `ExternalInterface.call()` method are passed along to the container as parameters of the function call.

```
// calls the external function "addNumbers"
// passing two parameters, and assigning that function's result
// to the variable "result"
var param1:uint = 3;
var param2:uint = 7;
var result:uint = ExternalInterface.call("addNumbers", param1, param2);
```

If the container is an HTML page, this method invokes the JavaScript function with the specified name, which must be defined in a `<script>` element in the containing HTML page. The return value of the JavaScript function is passed back to ActionScript.

```
<script language="JavaScript">
  // adds two numbers, and sends the result back to ActionScript
  function addNumbers(num1, num2)
  {
    return (num1 + num2);
  }
</script>
```

If the container is some other ActiveX container, this method causes the Flash Player ActiveX control to dispatch its `FlashCall` event. The specified function name and any parameters are serialized into an XML string by Flash Player. The container can access that information in the `request` property of the event object and use it to determine how to execute its own code. To return a value to ActionScript, the container code calls the ActiveX object's `SetReturnValue()` method, passing the result (serialized into an XML string) as a parameter of that method. For more information about the XML format used for this communication, see "The External API's XML format" on page 522.

Whether the container is a web browser or another ActiveX container, if the call fails or the container method does not specify a return value, `null` is returned. The `ExternalInterface.call()` method throws a SecurityError exception if the containing environment belongs to a security sandbox to which the calling code does not have access. You can work around this by setting an appropriate value for `allowScriptAccess` in the containing environment. For example, to change the value of `allowScriptAccess` in an HTML page, you would edit the appropriate attribute in the `<object>` and `<embed>` tags.

Calling ActionScript code from the container

A container can only call ActionScript code that's in a function—no other ActionScript code can be called by a container. To call an ActionScript function from the container application, you must do two things: register the function with the ExternalInterface class, and then call it from the container's code.

First, you must register your ActionScript function to indicate that it should be made available to the container. Use the `ExternalInterface.addCallback()` method, as follows:

```
function callMe(name:String):String
{
  return "busy signal =)";
}
ExternalInterface.addCallback("myFunction", callMe);
```

The `addCallback()` method takes two parameters. The first, a function name as a String, is the name by which the function will be known to the container. The second parameter is the actual ActionScript function that will be executed when the container calls the defined function name. Because these names are distinct, you can specify a function name that will be used by the container, even if the actual ActionScript function has a different name. This is especially useful if the function name is not known—for example, if an anonymous function is specified, or if the function to be called is determined at run time.

Once an ActionScript function has been registered with the ExternalInterface class, the container can actually call the function. How this is done varies according to the type of container. For example, in JavaScript code in a web browser, the ActionScript function is called using the registered function name as though it's a method of the Flash Player browser object (that is, a method of the JavaScript object representing the `<object>` or `<embed>` tag). In other words, parameters are passed and a result is returned as though a local function is being called.

```
<script language="JavaScript">
  // callResult gets the value "busy signal =)"
  var callResult = flashObject.myFunction("my name");
</script>
...
<object id="flashObject"...>
  ...
  <embed name="flashObject".../>
</object>
```

Alternatively, when calling a function in a SWF file running in an embedded ActiveX control, the registered function name and any parameters must be serialized into an XML-formatted string. Then the call is actually performed by calling the `CallFunction()` method of the ActiveX control with the XML string as a parameter.

In either case, the return value of the ActionScript function is passed back to the container code, either directly as a value when the caller is JavaScript code in a browser, or serialized as an XML-formatted string when the caller is an ActiveX container.

Example: Using the External API with a web page container

This example application demonstrates appropriate techniques for communicating between ActionScript and JavaScript in a web browser, in the context of an Instant Messaging application that allows a person to chat with him or herself (hence the name of the application: Introvert IM). Messages are sent between an HTML form in the web page and a SWF interface using the External API. The techniques demonstrated by this example include the following:

- Properly initiating communication by verifying that the browser is ready to communicate before setting up communication
- Checking whether the container supports the External API
- Calling JavaScript functions from ActionScript, passing parameters, and receiving values in response
- Making ActionScript methods available to be called by JavaScript, and performing those calls

The Introvert IM application files can be found in the Samples/IntrovertIM_HTML folder. The application consists of the following files:

File	Description
IntrovertIMApp.mxml	The main application file consisting of the MXML user interface.
com/example/programmingas3/introvertIM/IMManager.as	The class that establishes and manages communication between ActionScript and the container.
com/example/programmingas3/introvertIM/IMMessageEvent.as	Custom event type, dispatched by the IMManager class when a message is received from the container.
com/example/programmingas3/introvertIM/IMStatus.as	Enumeration whose values represent the different "availability" status values that can be selected in the application.
html-template/index.template.html	The template that is used to create the container HTML page for the application. This file contains all the JavaScript functions that make up the container part of the application.

Preparing for ActionScript-browser communication

One of the most common uses for the External API is to allow ActionScript applications to communicate with a web browser. Using the External API, ActionScript methods can call code written in JavaScript and vice versa. Because of the complexity of browsers and how they render pages internally, there is no way to guarantee that a SWF document will register its callbacks before the first JavaScript on the HTML page runs. For that reason, before calling functions in the SWF document from JavaScript, your SWF document should always call the HTML page to notify it that the SWF document is ready to accept connections.

For example, through a series of steps performed by the IMManager class, the Introvert IM determines whether the browser is ready for communication and prepares the SWF file for communication. The first step, determining when the browser is ready for communication, happens in the IMManager constructor, as follows:

```
public function IMManager(initialStatus:IMStatus)
{
  _status = initialStatus;

  // Check if the container is able to use the External API.
  if (ExternalInterface.available)
  {
    try
    {
      // This calls the isContainerReady() method, which in turn calls
      // the container to see if Flash Player has loaded and the container
      // is ready to receive calls from the SWF.
      var containerReady:Boolean = isContainerReady();
      if (containerReady)
      {
        // If the container is ready, register the SWF's functions.
        setupCallbacks();
      }
      else
      {
        // If the container is not ready, set up a Timer to call the
        // container at 100ms intervals. Once the container responds that
        // it's ready, the timer will be stopped.
        var readyTimer:Timer = new Timer(100);
        readyTimer.addEventListener(TimerEvent.TIMER, timerHandler);
        readyTimer.start();
      }
    }
    ...
  }
  else
  {
    trace("External interface is not available for this container.");
    }
}
```

First of all, the code checks whether the External API is even available in the current container using the `ExternalInterface.available` property. If so, it begins the process of setting up communication. Because security exceptions and other errors can occur when you attempt communication with an external application, the code is wrapped in a `try` block (the corresponding `catch` blocks were omitted from the listing for brevity).

The code next calls the `isContainerReady()` method, listed here:

```
private function isContainerReady():Boolean
{
    var result:Boolean = ExternalInterface.call("isReady");
    return result;
}
```

The `isContainerReady()` method in turn uses `ExternalInterface.call()` method to call the JavaScript function `isReady()`, as follows:

```
<script language="JavaScript">
<!--
// ------- Private vars -------
var jsReady = false;
...
// ------- functions called by ActionScript -------
// called to check if the page has initialized and JavaScript is available
function isReady()
{
    return jsReady;
}
...
// called by the onload event of the <body> tag
function pageInit()
{
    // Record that JavaScript is ready to go.
    jsReady = true;
}
...
//-->
</script>
```

The `isReady()` function simply returns the value of the `jsReady` variable. That variable is initially `false`; once the `onload` event of the web page has been triggered, the variable's value is changed to `true`. In other words, if ActionScript calls the `isReady()` function before the page is loaded, JavaScript returns `false` to `ExternalInterface.call("isReady")`, and consequently the ActionScript `isContainerReady()` method returns `false`. Once the page has loaded, the JavaScript `isReady()` function returns `true`, so the ActionScript `isContainerReady()` method also returns `true`.

Back in the IMManager constructor, one of two things happens depending on the readiness of the container. If `isContainerReady()` returns `true`, the code simply calls the `setupCallbacks()` method, which completes the process of setting up communication with JavaScript. On the other hand, if `isContainerReady()` returns `false`, the process is essentially put on hold. A Timer object is created and is told to call the `timerHandler()` method every 100 milliseconds, as follows:

```
private function timerHandler(event:TimerEvent):void
{
  // Check if the container is now ready.
  var isReady:Boolean = isContainerReady();
  if (isReady)
  {
    // If the container has become ready, we don't need to check anymore,
    // so stop the timer.
    Timer(event.target).stop();
    // Set up the ActionScript methods that will be available to be
    // called by the container.
    setupCallbacks();
  }
}
```

Each time the `timerHandler()` method gets called, it once again checks the result of the `isContainerReady()` method. Once the container is initialized, that method returns `true`. The code then stops the Timer and calls the `setupCallbacks()` method to finish the process of setting up communication with the browser.

Exposing ActionScript methods to JavaScript

As the previous example showed, once the code determines that the browser is ready, the `setupCallbacks()` method is called. This method prepares ActionScript to receive calls from JavaScript, as shown here:

```
private function setupCallbacks():void
{
  // Register the SWF client functions with the container
  ExternalInterface.addCallback("newMessage", newMessage);
  ExternalInterface.addCallback("getStatus", getStatus);
  // Notify the container that the SWF is ready to be called.
  ExternalInterface.call("setSWFIsReady");
}
```

The `setCallBacks()` method finishes the task of preparing for communication with the container by calling `ExternalInterface.addCallback()` to register the two methods that will be available to be called from JavaScript. In this code, the first parameter—the name by which the method is known to JavaScript (`"newMessage"` and `"getStatus"`)—is the same as the method's name in ActionScript. (In this case, there was no benefit to using different names, so the same name was reused for simplicity.) Finally, the `ExternalInterface.call()` method is used to call the JavaScript function `setSWFIsReady()`, which notifies the container that the ActionScript functions have been registered.

Communication from ActionScript to the browser

The Introvert IM application demonstrates a range of examples of calling JavaScript functions in the container page. In the simplest case (an example from the `setupCallbacks()` method), the JavaScript function `setSWFIsReady()` is called without passing any parameters or receiving a value in return:

```
ExternalInterface.call("setSWFIsReady");
```

In another example from the `isContainerReady()` method, ActionScript calls the `isReady()` function and receives a Boolean value in response:

```
var result:Boolean = ExternalInterface.call("isReady");
```

You can also pass parameters to JavaScript functions using the External API. For instance, consider the IMManager class's `sendMessage()` method, which is called when the user is sending a new message to his or her "conversation partner:"

```
public function sendMessage(message:String):void
{
  ExternalInterface.call("newMessage", message);
}
```

Once again, `ExternalInterface.call()` is used to call the designated JavaScript function, notifying the browser of the new message. In addition, the message itself is passed as an additional parameter to `ExternalInterface.call()`, and consequently it is passed as a parameter to the JavaScript function `newMessage()`.

Calling ActionScript code from JavaScript

Communication is supposed to be a two-way street, and the Introvert IM application is no exception. Not only does the Flash Player IM client call JavaScript to send messages, but the HTML form calls JavaScript code to send messages to and ask for information from the SWF file as well. For example, when the SWF file notifies the container that it has finished establishing contact and it's ready to communicate, the first thing the browser does is call the IMManager class's `getStatus()` method to retrieve the initial user availability status from the SWF IM client. This is done in the web page, in the `updateStatus()` function, as follows:

```
<script language="JavaScript">
...
function updateStatus()
{
  if (swfReady)
  {
    var currentStatus = getSWF("IntrovertIMApp").getStatus();
    document.forms["imForm"].status.value = currentStatus;
  }
}
...
</script>
```

The code checks the value of the `swfReady` variable, which tracks whether the SWF file has notified the browser that it has registered its methods with the ExternalInterface class. If the SWF file is ready to receive communication, the next line (`var currentStatus` = ...) actually calls the `getStatus()` method in the IMManager class. Three things happen in this line of code:

- The `getSWF()` JavaScript function is called, returning a reference to the JavaScript object representing the SWF file. The parameter passed to `getSWF()` determines which browser object is returned in case there is more than one SWF file in an HTML page. The value passed to that parameter must match the `id` attribute of the `<object>` tag and `name` attribute of the `<embed>` tag used to embed the SWF file.
- Using the reference to the SWF file, the `getStatus()` method is called as though it's a method of the SWF object. In this case the function name "`getStatus`" is used because that's the name under which the ActionScript function is registered using `ExternalInterface.addCallback()`.

- The `getStatus()` ActionScript method returns a value, and that value is assigned to the `currentStatus` variable, which is then assigned as the content (the `value` property) of the `status` text field.

NOTE

If you're following along in the code, you've probably noticed that in the source code for the `updateStatus()` function, the line of code that calls the `getSWF()` function, is actually written as follows:

```
var currentStatus = getSWF("${application}").getStatus();
```

The `${application}` text is a placeholder in the HTML page template; when Adobe Flex Builder 2 generates the actual HTML page for the application, this placeholder text is replaced by the same text that is used as the `<object>` tag's `id` attribute and the `<embed>` tag's `name` attribute (`IntrovertIMApp` in the example). That is the value that is expected by the `getSWF()` function.

The `sendMessage()` JavaScript function demonstrates passing a parameter to an ActionScript function. (`sendMessage()` is the function that is called when the user presses the Send button on the HTML page.)

```
<script language="JavaScript">
...
function sendMessage(message)
{
  if (swfReady)
  {
    ...
    getSWF("IntrovertIMApp").newMessage(message);
  }
}
...
</script>
```

The `newMessage()` ActionScript method expects one parameter, so the JavaScript `message` variable gets passed to ActionScript by using it as a parameter in the `newMessage()` method call in the JavaScript code.

Detecting the browser type

Because of differences in how browsers access content, it's important to always use JavaScript to detect which browser the user is running and to access the movie according to the browser-specific syntax, using the window or document object, as shown in the `getSWF()` JavaScript function in this example:

```
<script language="JavaScript">
...
function getSWF(movieName)
{
  if (navigator.appName.indexOf("Microsoft") != -1)
{
    return window[movieName];
  }
  else
  {
    return document[movieName];
  }
}
...
</script>
```

If your script does not detect the user's browser type, the user might see unexpected behavior when playing SWF files in an HTML container.

Example: Using the External API with an ActiveX container

This example demonstrates using the External API to communicate between ActionScript and a desktop application that uses the ActiveX control. The example reuses the Introvert IM application, including the ActionScript code and even the same SWF file, and therefore does not describe the use of the External API in ActionScript. Familiarity with the preceding example will be helpful in understanding this one.

The desktop application in this example is written in C# using Microsoft Visual Studio .NET. The focus of the discussion is the specific techniques for working with the External API using the ActiveX control. This example demonstrates the following:

- Calling ActionScript functions from a desktop application hosting the Flash Player ActiveX control
- Receiving function calls from ActionScript and processing them in an ActiveX container
- Using a proxy class to hide the details of the serialized XML format that Flash Player uses for messages sent to an ActiveX container

The Introvert IM C# files can be found in the Samples/IntrovertIM_CSharp folder. The application consists of the following files:

File	Description
AppForm.cs	The main application file with the C# Windows Forms interface.
bin/Debug/IntrovertIMApp.swf	The SWF file loaded by the application.
ExternalInterfaceProxy/ ExternalInterfaceProxy.cs	The class that serves as a wrapper around the ActiveX control for External Interface communication. It provides mechanisms for calling and receiving calls from ActionScript.
ExternalInterfaceProxy/ ExternalInterfaceSerializer.cs	The class that performs the task of converting Flash Player's XML format messages to .NET objects.
ExternalInterfaceProxy/ ExternalInterfaceEventArgs.cs	This file defines two C# types (classes): a custom delegate, and an event arguments class, which are used by the ExternalInterfaceProxy class to notify a listener of a function call from ActionScript.
ExternalInterfaceProxy/ ExternalInterfaceCall.cs	This class is a value object representing a function call from ActionScript to the ActiveX container, with properties for the function name and parameters.
bin/Debug/IntrovertIMApp.swf	The SWF file loaded by the application.
obj/AxInterop.ShockwaveFlashObjects.dll, obj/Interop.ShockwaveFlashObjects.dll	Wrapper assemblies created by Visual Studio .NET that are required to access the Flash Player (Shockwave® Flash) ActiveX control from managed code.

Overview of the Introvert IM C# Application

This example application represents two instant-messaging client programs (one within a SWF file and another built with Windows Forms) that communicate with each other. The user interface includes an instance of the Shockwave® Flash ActiveX control, within which the SWF file containing the ActionScript IM client is loaded. The interface also includes several text fields that make up the Windows Forms IM client: a field for entering messages (`MessageText`), another that displays the transcript of the messages sent between the clients (`Transcript`), and a third (`Status`) that displays the availability status as set in the SWF IM client.

Including the Shockwave Flash ActiveX control

To include the Shockwave Flash ActiveX control in your own Windows Forms application, you must first add it to the Microsoft Visual Studio Toolbox.

To add the control to the toolbox:

1. Open the Visual Studio Toolbox.
2. Right-click the Windows Forms section in Visual Studio 2003 or any section in Visual Studio 2005. From the context menu select Add/Remove Items in Visual Studio 2003 (Choose Items... in Visual Studio 2005).

   This opens the Customize Toolbox (2003)/Choose Toolbox Items (2005) dialog box.
3. Select the COM Components tab, which lists all of the available COM components on your computer, including the Flash Player ActiveX control.
4. Scroll to Shockwave Flash Object and select it.

   If this item is not listed, make sure that the Flash Player ActiveX control is installed on your system.

Understanding ActionScript to ActiveX container communication

Communication using the External API with an ActiveX container application works like communication with a web browser, with one important difference. As described earlier, when ActionScript communicates with a web browser, as far as the developer is concerned, the functions are called directly; the details of how the function calls and responses are formatted to pass them between the player and browser are hidden. However, when the External API is used to communicate with an ActiveX container application, Flash Player sends messages (function calls and return values) to the application in a specific XML format, and it expects function calls and return values from the container application to use the same XML format. The developer of the ActiveX container application must write code that understands and can create function calls and responses in the appropriate format.

The Introvert IM C# example includes a set of classes that allow you to avoid formatting messages; instead, you can work with standard data types when calling ActionScript functions and receiving function calls from ActionScript. The ExternalInterfaceProxy class, together with other helper classes, provides this functionality, and can be reused in any .NET project to facilitate External API communication.

The following sections of code, excerpted from the main application form (AppForm.cs), demonstrate the simplified interaction that is achieved by using the ExternalInterfaceProxy class:

```
public class AppForm : System.Windows.Forms.Form
{
  ...
  private ExternalInterfaceProxy proxy;
  ...
  public AppForm()
  {
    ...
    // Register this app to receive notification when the proxy receives
    // a call from ActionScript.
    proxy = new ExternalInterfaceProxy(IntrovertIMApp);
    proxy.ExternalInterfaceCall += new
  ExternalInterfaceCallEventHandler(proxy_ExternalInterfaceCall);
    ...
  }
  ...
```

The application declares and creates an ExternalInterfaceProxy instance named `proxy`, passing in a reference to the Shockwave Flash ActiveX control that is in the user interface (`IntrovertIMApp`). Next, the code registers the `proxy_ExternalInterfaceCall()` method to receive the proxy's `ExternalInterfaceCall` event. This event is dispatched by the ExternalInterfaceProxy class when a function call comes from Flash Player. Subscribing to this event is the way the C# code receives and responds to function calls from ActionScript.

When a function call comes from ActionScript, the ExternalInterfaceProxy instance (`proxy`) receives the call, converts it from XML format, and notifies the objects that are listeners for the proxy's ExternalInterfaceCall event. In the case of the AppForm class, the `proxy_ExternalInterfaceCall()` method handles that event, as follows:

```
/// <summary>
/// Called by the proxy when an ActionScript ExternalInterface call
/// is made by the SWF
/// </summary>
private object proxy_ExternalInterfaceCall(object sender,
ExternalInterfaceCallEventArgs e)
{
  switch (e.FunctionCall.FunctionName)
  {
    case "isReady":
      return isReady();
    case "setSWFIsReady":
      setSWFIsReady();
      return null;
    case "newMessage":
      newMessage((string)e.FunctionCall.Arguments[0]);
      return null;
    case "statusChange":
      statusChange();
      return null;
    default:
      return null;
  }
}
...
```

The method gets passed an ExternalInterfaceCallEventArgs instance, named `e` in this example. That object, in turn, has a `FunctionCall` property that is an instance of the ExternalInterfaceCall class.

An ExternalInterfaceCall instance is a simple value object with two properties. The `FunctionName` property contains the function name specified in the ActionScript `ExternalInterface.Call()` statement. If any parameters are added in ActionScript, those parameters are included in the ExternalInterfaceCall object's `Arguments` property. In this case, the method that handles the event is simply a `switch` statement that acts like a traffic manager. The value of the `FunctionName` property (`e.FunctionCall.FunctionName`) determines which method of the AppForm class is called.

The branches of the `switch` statement in the previous code listing demonstrate common method call scenarios. For instance, any method must return a value to ActionScript (for example, the `isReady()` method call) or else should return `null` (as seen in the other method calls). Accessing parameters passed from ActionScript is demonstrated in the `newMessage()` method call (which passes along a parameter `e.FunctionCall.Arguments[0]`, the first element of the `Arguments` array).

Calling an ActionScript function from C# using the ExternalInterfaceProxy class is even more straightforward than receiving a function call from ActionScript. To call an ActionScript function, you use the ExternalInterfaceProxy instance's `Call()` method, as follows:

```
  /// <summary>
  /// Called when the "Send" button is pressed; the value in the
  /// MessageText text field is passed in as a parameter.
  /// </summary>
  /// <param name="message">The message to send.</param>
  private void sendMessage(string message)
  {
    if (swfReady)
    {
      ...
      // Call the newMessage function in ActionScript.
      proxy.Call("newMessage", message);
    }
  }
  ...
  /// <summary>
  /// Call the ActionScript function to get the current "availability"
  /// status and write it into the text field.
  /// </summary>
  private void updateStatus()
  {
    Status.Text = (string)proxy.Call("getStatus");
  }
  ...
}
```

As this example shows, the ExternalInterfaceProxy class's `Call()` method is very similar to its ActionScript counterpart, `ExternalInterface.Call()`. The first parameter is a string, the name of the function to call. Any additional parameters (not shown here) are passed along to the ActionScript function. If the ActionScript function returns a value, that value is returned by the `Call()` method (as seen in the previous example).

Inside the ExternalInterfaceProxy class

Using a proxy wrapper around the ActiveX control may not always be practical, or you may wish to write your own proxy class (for instance, in a different programming language or targeting a different platform). Although not all the details of creating a proxy will be explained here, it is instructive to understand the inner workings of the proxy class in this example.

You use the Shockwave Flash ActiveX control's `CallFunction()` method to call an ActionScript function from the ActiveX container using the External API. This is shown in this extract from the ExternalInterfaceProxy class's `Call()` method:

```
// Call an ActionScript function on the SWF in "_flashControl",
// which is a Shockwave Flash ActiveX control.
string response = _flashControl.CallFunction(request);
```

In this code excerpt, `_flashControl` is the Shockwave Flash ActiveX control. ActionScript function calls are made using the `CallFunction()` method. That method takes one parameter (`request` in the example), which is a string containing XML-formatted instructions including the name of the ActionScript function to call and any parameters. Any value returned from ActionScript is encoded as an XML-formatted string and sent back as the return value of the `CallFunction()` call. In this example, that XML string is stored in the `response` variable.

Receiving a function call from ActionScript is a multistep process. Function calls from ActionScript cause the Shockwave Flash ActiveX control to dispatch its FlashCall event, so a class (such as the ExternalInterfaceProxy class) that intends to receive calls from a SWF file needs to define a handler for that event. In the ExternalInterfaceProxy class, the event handler function is named `_flashControl_FlashCall()`, and it is registered to listen for the event in the class constructor, as follows:

```
private AxShockwaveFlash _flashControl;

public ExternalInterfaceProxy(AxShockwaveFlash flashControl)
{
  _flashControl = flashControl;
  _flashControl.FlashCall += new
  _IShockwaveFlashEvents_FlashCallEventHandler(_flashControl_FlashCall);
}
...
private void _flashControl_FlashCall(object sender,
  _IShockwaveFlashEvents_FlashCallEvent e)
{
  // Use the event object's request property ("e.request")
  // to execute some action.
  ...
  // Return a value to ActionScript;
  // the returned value must first be encoded as an XML-formatted string.
  _flashControl.SetReturnValue(encodedResponse);
}
```

The event object (`e`) has a `request` property (`e.request`) that is a string containing information about the function call, such as the function name and parameters, in XML format. This information can be used by the container to determine what code to execute. In the ExternalInterfaceProxy class, the request is converted from XML format to an ExternalInterfaceCall object, which provides the same information in a more accessible form. The ActiveX control's `SetReturnValue()` method is used to return a function result to the ActionScript caller; once again, the result parameter must be encoded in the XML format used by the External API.

The External API's XML format

Communication between ActionScript and an application hosting the Shockwave Flash ActiveX control uses a specific XML format to encode function calls and values. In the Introvert IM C# example, the ExternalInterfaceProxy class makes it possible for the code in the application form to ignore this format and operate directly on the values. In order to accomplish this, the ExternalInterfaceProxy class uses the methods of the ExternalInterfaceSerializer class to actually translate the XML messages into .NET objects. The ExternalInterfaceSerializer class has four public methods:

- `EncodeInvoke()` Encodes a function name and a C# ArrayList of arguments into the appropriate XML format.
- `EncodeResult()` Encodes a result value into the appropriate XML format.
- `DecodeInvoke()` Decodes a function call from ActionScript. The FlashCall event object's `request` property is passed to the `DecodeInvoke()` method, and it translates the call into an ExternalInterfaceCall object.
- `DecodeResult()` Decodes the XML received as the result of calling an ActionScript function.

There are two parts to the XML format used by the External API. One format is used to represent function calls. Another format is used to represent individual values; this format is used for parameters in functions as well as function return values. The XML format for function calls is used for calls to and from ActionScript. For a function call from ActionScript, Flash Player passes the XML to the container; for a call from the container, Flash Player expects the container application to pass it an XML string in this format. The following XML fragment shows an example XML-formatted function call:

```
<invoke name="functionName" returntype="xml">
  <arguments>
    ... (individual argument values)
  </arguments>
</invoke>
```

The root node is the `<invoke>` node. It has two attributes: `name` indicates the name of the function to call, and `returntype` is always `xml`. If the function call includes parameters, the `<invoke>` node has a child `<arguments>` node, whose child nodes will be the parameter values formatted using the individual value format explained next.

Individual values, including function parameters and function return values, use a formatting scheme that includes data type information in addition to the actual values:

ActionScript class/value	**C# class/ value**	**Format**	**Comments**
null	null	`<null/>`	
Boolean `true`	bool true	`<true/>`	
Boolean `false`	bool false	`<false/>`	
String	string	`<string>string value</string>`	

Number, int, uint	single, double, int, uint	`<number>27.5</number>`
		`<number>-12</number>`	

Array (elements can be mixed types)	A collection that allows mixed-type elements, such as ArrayList or object[]	`<array>`
		   `<property id="0">`
		      `<number>27.5</number>`
		   `</property>`
		   `<property id="1">`
		      `<string>Hello there!</string>`
		   `</property>`
		   `...`
		`</array>`
			The `<property>` node defines individual elements, and the `id` attribute is the numeric, zero-based index.

Object	A dictionary with string keys and object values, such as a HashTable with string keys	`<object>`
		   `<property id="name">`
		      `<string>John Doe</string>`
		   `</property>`
		   `<property id="age">`
		      `<string>33</string>`
		   `</property>`
		   `...`
		`</object>`
			The `<property>` node defines individual properties, and the `id` attribute is the property name (a string).

Other built-in or custom classes		`<null/>`
		   `or`
		`<object></object>`
			ActionScript encodes other objects as null or as an empty object. In either case any property values are lost.