Public Beta 1

Programming ActionScript 3.0

Acknowledgments

Project Management:

Writing:

Editing:

Production Management:

Media Design and Production:

Special thanks to

First Edition: January 2006

Adobe Systems Incorporated
601 Townsend St.
San Francisco, CA 94103

Contents

PART 1

Overview of ActionScript Programming

1

This book targets developers who have some programming experience. Part 1 of the book describes how to implement programming concepts in ActionScript.

For information on basic programming concepts, see Learning ActionScript instead.

The following chapters are included:

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

CHAPTER 1

Introduction

1

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

What Is ActionScript?

Why Use ActionScript?

Documentation Map

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Links to other docs

Developer Center

ActionScript in Context

Based on ECMAScript standards

Why ECMAScript compliance is important

One language used by many products: Flash, Flex, Remoting, Comm Server

What's New in ActionScript 3.0

New features

Language changes

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Performance improvements

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

CHAPTER 2

Getting Started with ActionScript

2

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

A basic ActionScript development process

Designing your ActionScript Application

Creating ActionScript Code

Options for organizing your code

Storing code on frames in a Flash timeline

Embedding code in Flex MXML files

Storing code in separate ActionScript files

Creating the HelloWorld class

Creating a Flash or Flex application that uses your ActionScript code

Using external ActionScript files in your application

Referencing the HelloWorld class

Publishing and testing your ActionScript application

Enhancing the Hello World application

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 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.

Language Overview

Objects lie at the heart of the ActionScript 3.0 language—they are its fundamental building blocks. Every variable you create, every function you write, and every class instance you instantiate 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. ECMAScript, the standard upon which ActionScript 3.0 is based, defines objects in a similar, but slightly different way. In ECMAScript, 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 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 Language Reference, for example, uses the term *properties* to mean variables or getter/setter properties, and 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 and 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, but different, 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 of a package. Packages let you organize your class files, and namespaces let you manage the visibility of 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 users, 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. This means that unlike Java, ActionScript 3.0 does not allow you to declare 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 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. 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 contains the legacy XML parser that was used in previous versions of ActionScript. One reason 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) 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 users 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.

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

Most of the Flash Player API is organized under the `flash` package. For example, `flash.display` contains the display list API and `flash.events` contains the new event model. A detailed discussion of the Flash Player API packages can be found in Part III of this book. For more information, see "Flash Player APIs" on page 195.

Packages are useful for organizing your code and for preventing name conflicts. You should not confuse the concept of packages with the concept of class inheritance, as they are unrelated. 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 `tutorial.samples`, you must use one of the following import statements before using the `SampleCode` class:

```
import tutorial.samples.*;
```

or

```
import tutorial.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.

After you have properly imported the class or package, you can use either the fully qualified name of the class (`tutorial.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:tutorial.samples.SampleCode = new
  tutorial.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 of the package, you must declare that class to be `public`. For example, the following package contains two classes, SampleCode and CodeFormatter.

```
package tutorial.samples {
  public class SampleCode {}
  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 `sample` package itself. If you attempt to access the CodeFormatter class outside of the `tutorial.samples` package, you will generate an error.

```
import tutorial.samples.SampleCode;
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 and place them in separate source files. 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, you will have a name conflict when referring to the SampleCode class.

```
import tutorial.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.

```
var mySample:tutorial.samples.SampleCode = new
  tutorial.samples.SampleCode();
var mySample: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. The `public`, `private`, `protected` and `internal` access control specifiers can be thought of 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, though 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 identifiers are not preceded by a namespace qualifier, they are qualified by the default `internal` namespace, which means they are visible only to callers in the same package. 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, you can then reference the namespace 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.

```
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.util package. As shown 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.util package and would not be visible to any code outside the package.

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

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

```
package flash.util {
    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.

```
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 `use namespace` "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

A real-world example of a namespace that is used to prevent name conflicts can be found in the flash.util.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 `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.util package in a manner similar to the following:

```
package flash.util {

  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.util.trace;
```

```
  import flash.util.Proxy;
  import flash.util.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(); // Output: 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 Proxy. 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.

In the following example, a user-defined namespace is used 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 this technique. 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 into 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
  import flash.util.trace;

  use namespace myInternal;

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

      Helper.someTask();
      trace(Helper.lastCalled);   // Output: [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;
```

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 of `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:

```
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 `int` and `String`, but also when creating an array or instantiating an instance of a class. The following example shows an array 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 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); // output: Global
}
scopeTest();
trace (strGlobal); // output: 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 of 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`, 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); // Output: Local
}
scopeTest();
trace (str1); // Output: 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 of 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() of the num variable happens before the num variable is declared:

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

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

```
num = 5;
trace(num); // Output: 5
var num:Number = 10;
trace(num); // Output: 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 value is untyped. If you also do not initialize an untyped variable with a value, then 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 of 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 called `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 of 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:

```
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 runtime. 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 runtime. As a dynamically typed language, ActionScript 3.0 has runtime 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 runtime, but in standard mode, type checking occurs only at runtime.

Dynamically typed languages offer tremendous flexibility when structuring your code, but at the cost of allowing type errors to manifest at runtime. 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 FlexBuilder 2.0 is set to run in strict mode. You can disable strict mode in FlexBuilder 2.0 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-hand 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");
```

Runtime type checking

Runtime 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 runtime type checking by Flash Player results in a runtime error. The following example shows a function named `typeTest` that expects an Array argument but is passed a value of `3`. This causes a runtime 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);
// Runtime error in ActionScript 3.0 standard mode
```

There may also be situations where you get a runtime type error even when 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 runtime. 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 runtime error because it compares the runtime 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);
// Runtime error in ActionScript 3.0
```

Runtime type checking also allows more flexible use of inheritance than does compile time checking. By deferring type checking to runtime, 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`.

```
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 show 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 no longer provides 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);                  // output: true
trace (mySprite is DisplayObject);           // output: true
trace (mySprite is IEventDispatcher);        // output: 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 implements 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, but returns `false` when used to test whether `mySprite` is an instance of the DisplayObect class or implements the IEventDispatcher interface.

```
trace (mySprite instanceof Sprite);          // output: true
trace (mySprite instanceof DisplayObject);   // output: false
trace (mySprite instanceof IEventDispatcher); // output: 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 and IEventDispatcher data types.

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

The right-hand operand used with the `as` operator must be a data type. An attempt to use an expression other than a data type as the right-hand operand will result in an error.

A common usage of the `as` operator is to assign the result of an `as` operation to a variable of the same type. For example, the following code takes a zip code from an online form that is a String value and assigns the result of an `as` operation to a variable of type Number. The `num` variable is assigned the numeric value `94103` because Flash Player converts the string to a numeric value per the implicit conversion rules of ActionScript 3.0.

```
var zipCode:String = "94103"
var num:Number = zipCode as Number;
trace(num); // output: 94103
```

The main benefit of using the `as` operator is that it should guarantee that the result of the `as` operation will either be the value of the left-hand expression or `null`. In the zip code example, however, the variable `num` is a primitive data type, and none of the primitive data types contain the value `null`. In these situations, Flash Player converts the value `null` into the default value of primitive data type. In the case of the Number data type, the default value is `NaN`. For example, if the `zipCode` variable in the previous example is a nine digit zip code with a hyphen, the conversion will fail. The `as` operator attempts to return `null`, because the implicit conversion failed, but the `num` variable cannot accept a `null` value, so the default value of the Number data type, `NaN`, is used instead.

```
var zipCode:String = "94103-1734"
var num:Number = zipCode as Number;
trace(num); // output: NaN
```

Dynamic classes

A *dynamic* class defines an object that can be altered at runtime 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 runtime. 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); // Output: testing 3
```

Properties that you add to an instance of a dynamic Class are runtime entities, so any type checking is done at runtime. You cannot add a type annotation to property 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 (myProtean.aString, myProtean.aNumber);
}
myProtean.traceProtean(); // Output: 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 fully qualified. 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); // Output: undefined
  trace(myProtean.publicGreeting);  // Output: 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`. For more information about complex data types, see Chapter 5, "Core Classes Overview" on page 119.

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 the value `null` to variables of type Boolean, Number, int, or uint.

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 fifty two 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 is 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, so 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 the 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 its 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 runtime. 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); // Output: 1
```

Implicit conversions

Implicit conversions happen at runtime in a number of contexts:

- in assignment statements;
- when values are passed as function arguments;
- when values are returned from functions;
- in `as` expressions; and
- 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 either 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

The use of explicit conversions, or casting, is helpful when compiling 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 runtime. 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:

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

In strict mode, the compiler tests for type compatibility with the `is` operator. If the `is` operator would return `false`, the compiler generates an error.

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); // output: 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // output: 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 (e.g. `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 truncated. For example, the following string values can be cast into numbers.

```
trace(uint("5"));     // Output: 5
trace(uint("-5"));    // Output: -5
trace(uint(" 27 "));  // Output: 27
trace(uint("3.7"));   // Output: 3
trace(int("3.7"));    // Output: 3
trace(int("0x1A"));   // Output: 26
trace(Number("3.7")); // Output: 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"));    // Output: 0
trace(uint("ten"));   // Output: 0
trace(uint("17 63")); // Output: 0
```

In ActionScript 3.0, the `Number()` function no longer supports octal, or base 8, numbers. If you supply a string with a leading `0` 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 output: 44
// ActionScript 2.0 output: 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); // Output: 4294967293

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

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

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

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. 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 (`""`), and `true` otherwise.

```
var str1:String;       // uninitialized string is null
trace(Boolean(str1));  // output: false

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

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

Casting to Boolean from an instance of the Object class returns `false` if the instance is `null`, and `true` otherwise.

```
var myObj:Object;      // uninitialized object is null
trace(Boolean(myObj)); // output: false

myObj = new Object();  // instantiate
trace(Boolean(myObj)); // output: 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 runtime.

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

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 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)); // output: 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 (output shows result for Pacific Daylight Time).

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

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;
  pubic function method1 () {}
}
```

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

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

Dot syntax can also be used when defining 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 character (`,`) 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 any ActionScript core class.

```
// 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 character (`,`) 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.

```
// using 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};
```

Semicolons

The semicolon character (;) can be used to terminate a statement. Alternatively, you can omit the semicolon character and 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

Parentheses (()) can be used in three ways in ActionScript 3.0. First, parentheses can be used 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);    // output: 14
trace ( (2 + 3) * 4); // output: 20
```

Second, parentheses can be used 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)); // output: 7
```

Third, parentheses can be used 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"); // output: hello
```

Comments

ActionScript 3.0 code supports two types of comments: single line comments and multi-line comments. These commenting mechanisms are similar to the commenting mechanisms available 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; // This is a single line comment
```

Multi-line comments begin with a forward slash and asterisk (/*) and end with an asterisk and forward slash (*/).

```
/* This is multi-line 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		

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); // Output: 0
trace(a.MAXIMUM); // Output: 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 runtime error will occur.

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

var a:A = new A();
a["MAXIMUM"] = 10 // runtime 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 is an operator that 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, then the addition (+) operator returns the sum of the values. If both operands are strings, then 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);     // Output: 10
trace ("5" + "5"); // Output: 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);  // Output: -3
trace(7-2); // Output: 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);   // output: false
trace ((3 > 2) < 1); // output: 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); // Output: 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 from the previous example.

```
trace (3 > (2 < 1)); // output: 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); // Output: 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. An operator has higher precedence than an operator appearing below it in the table.

Group	Operators
Primary	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Postfix	`x++ x--`
Unary	`++x --x + - ! T(x) typeof void`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >> >>>`
Relational	`< > <= >= instanceof in 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 of the operators in this table have equal precedence.

Operator	Operation performed
`[]`	Array initialization
`{x:y}`	Object initialization
`()`	Grouping
`f(x)`	Function called
`new`	Constructor called
`x.y x[y]`	Property access
`<></>`	XMLList initialization (E4X)
`@`	Attribute access (E4X)
`::`	Name qualifier (E4X)
`..`	Descendant accessor (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++); // Output: 0
trace (xNum);   // Output: 1
```

All of the operators in this table have equal precedence.

Operator	Operation performed
`++`	Increment (postfix)
`--`	Decrement (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 the 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); // Output: 1
trace (xNum);   // Output: 1
```

All of the operators in this table have equal precedence.

Operator	Operation performed
`++`	Increment (prefix)
`--`	Decrement (prefix)
`+`	Unary +
`!`	Unary - (negation)
`T(x)`	Type cast
`typeof`	Returns type information
`void`	Returns undefined value

Multiplicative operators

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

All of the operators in this table have equal precedence.

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

Additive operators

The additive operators take two operands and perform addition or subtraction calculations. The operators in this 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 of the operators in this 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 returns a Boolean value. All of the operators in this table have equal precedence.

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

Equality operators

The equality operators take two operands, compares their values, and returns a Boolean value. All of the operators in this 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 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 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 take three operands. The conditional operator is a short-hand 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 of the operators in this 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()` statement if it does, or generates a different `trace()` statement 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");
}
```

switch

The `switch` statement is useful if you have several execution paths. 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");
}
```

Looping

Looping statements allow you to perform a specific block of code repeatedly using a series of values or variables.

for

The `for` loop allows you to iterate over a variable for a specific range of values. You must supply three expressions to 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 will appear in a 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 custom class, unless the class is a dynamic class. Even with instances of dynamic classes, you will only be able to iterate through properties that are added dynamically.

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
```

Arrays

Arrays allow you to store multiple values in a single data structure. Arrays can be simple indexed arrays that store values using fixed ordinal integer indexes. Arrays can be more complex associative arrays that store values using arbitrary keys. Arrays can also be multi-dimensional arrays that contain elements which are themselves arrays.

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 one for each subsequent element added to the array. You can create an indexed array by calling the Array 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
```

Associative arrays

Associative arrays 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. 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.

There are two ways to create associative arrays in ActionScript 3.0. The first way is to use the Object constructor, which has the key 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 `MonitorArray`, 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:

```
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, then you cannot use an object literal to initialize the array. The following example creates an associative array named `monitorArray` 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 to 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.

Implementing enumerations as associative arrays

Enumerations are custom data types that programmers create to encapsulate a small set of values. For example, a programmer may want a variable to hold a specific day of the week. The programmer could simply use a string or an integer value to represent each day, but both of these options have disadvantages. The use of integers makes the code hard to read and maintain, but facilitates looping through the days of the week. The use of strings makes looping difficult, but makes the code more readable. Enumerations provide an elegant solution that stores both a string and an integer for each day of the week. This allows looping and makes the code easy to read.

ActionScript 3.0 does not support a specific enumeration facility, as does C++ with its `enum` keyword and Java with its Enumeration interface. You can achieve similar functionality in ActionScript, however, by using associative arrays. For example, the following code creates an associative array that contains each day of the week.

```
var Days:Object = {MON:0, TUE:1, WED:2, THU:3, FRI:4, SAT:5, SUN:6};
```

After creating a `Days` object, you can use its keys to assign values to variables:

```
var today:uint = Days.FRI;
```

You can also loop through the days of the week with a `for...in` loop:

```
for (var d:String in Days) {
  if (today == Days[d]) {
    trace ("Today is " + d)
  }
}
// Output: Today is FRI
```

The advantage of using a `for..in` loop instead of a `for` loop is that a `for...in` loop gives you access to both the enumeration key and its value. A `for` loop would only give you access to the integer value of each key and value pair.

Multi-dimensional arrays

Multi-dimensional 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 multi-dimensional 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 multi-dimensional arrays. The following examples use either two indexed arrays or an associative array of indexed arrays. The two other combinations are left as exercises for the reader.

Two indexed arrays

When two indexed arrays are used, 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 multi-dimensional array uses two indexed arrays to track task lists for each day of the week. The first array, `masterTaskList`, is created using the Array 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. These 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"];
```

To 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, 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
```

Two indexed arrays with enumeration

You can use an enumeration to make the elements easier to read so that at least one set of indices can be referred to by name instead of by number. For example, you can use an associative array of days so that each day can be accessed by an enumeration instead of by a number:

```
var Days:Object = {MON:0, TUE:1, WED:2, THU:3, FRI:4, SAT:5, SUN:6};
var masterTaskList:Array = new Array();
masterTaskList[Days.MON] = ["wash dishes", "take out trash"];
masterTaskList[Days.TUE] = ["wash dishes", "pay bills"];
masterTaskList[Days.WED] = ["wash dishes", "dentist", "wash dog"];
masterTaskList[Days.THU] = ["wash dishes"];
masterTaskList[Days.FRI] = ["wash dishes", "clean house"];
masterTaskList[Days.SAT] = ["wash dishes", "wash car", "pay rent"];
masterTaskList[Days.SUN] = ["mow lawn", "fix chair"];
```

NOTEthe example uses abbreviations for each day for brevity. You could choose to use the full name of each day instead.

You can access an item on any of the task lists using bracket notation. For example, the following code retrieves the second task from Wednesday's list, and the first task from Sunday's list.

```
trace(masterTaskList[Days.WED][1]); // Output: dentist
trace(masterTaskList[Days.SUN][0]); // Output: mow lawn
```

Associative array with an indexed array

To make the individual arrays even easier to access, you can use an associative array for the days of the week, and use indexed arrays for the task lists. Using an associative array not only allows you to avoid the use of an enumeration object, but also allows you to use dot syntax when referring to a particular 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"];
```

The use of dot syntax makes the code more readable by allowing you to avoid the use of multiple sets of brackets:

```
trace(masterTaskList.Wednesday[1]); // Output: dentist
trace(masterTaskList.Sunday[0]);    // Output: mow lawn
```

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, for example, the `class` keyword does not exist, so "classes" are 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. While basic function definition and invocation should not present a challenge to experienced programmers, some of the more advanced features of ActionScript functions will most likely require explanation.

Basic 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 in the flash.util package of 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 literal. The choice between the two techniques depends on whether you prefer a more static or dynamic programming style. The use of function statements is more conducive to static, or strict mode, programming, while the use of function literals is more suitable for dynamic, or standard mode, programming.

Function statement

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 places the parameter into a `trace()` statement and invokes the function using the string `hello` as a parameter:

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

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

Function Literal

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

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

- the function name;
- the colon operator (`:`);
- the `Function` keyword used 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 literal:

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

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

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

```
};
traceArray[0]("hello");
```

Choosing between statements and literals

As a general rule, use function statements unless specific circumstances call for the use of function literals. Function statements are not only less verbose than function literals, but also provide a more consistent experience between strict mode and standard mode.

Function statements are easier to read than assignment statements that contain function literals. Using function statements makes your code more concise and are less confusing than function literals, which require use of both `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 literal. For example, the following code defines a class named Example with two methods: `methodLiteral()`, which is declared with a function literal, and `methodStatement()`, which is declared with a function statement. In strict mode, you cannot use dot syntax to invoke the `methodLiteral()` method.

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

myExample.methodLiteral();   // Error in strict mode
myExample.methodStatement(); // Okay in strict and standard modes
```

The use of function literals is considered better suited to programming that focuses more on runtime, or dynamic, behavior. If you prefer to use strict mode, but also need to call a method declared with a function literal, 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, attempts to access an undefined property on an instance of a dynamic class are not caught at compile time.

There are some circumstances in which function literals are useful. One common usage of function literals is for functions that are used only once, then discarded. Another usage, albeit much less common, is for attaching a function to a prototype property. For more information, see "The prototype object" on page 88.

There are two subtle differences between function statements and function literals that you should take into account when choosing which type to use. The first difference is that function literals do not exist independently as objects with regard to memory management and garbage collection. In other words, when you assign a function literal to another object, such as an array element or an object property, you create the only reference to that function literal in your code. If the array or object to which your function literal is attached goes out of scope or is otherwise no longer available, you will no longer have access to the function literal. If the array or object is deleted, then the memory that the function literal 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 literal, once the property to which the literal is assigned is deleted, the function is no longer available. The class Test is dynamic, which means that you can add a property named `literal` that holds a function literal. The `literal()` function can be called with the dot operator, but once the literal property is deleted, the function is no longer accessible.

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

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

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

```
statementTest(); // Output: statementTest

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

Function literals are not available before they are defined, so the following code results in a runtime error:

```
literalTest(); // Runtime error

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

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);
}
```

Note that the return statement terminates the function, so that any statements below a return statement will not be executed:

```
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 only available 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 "8.5";
  }
  function getProductName():String {
    return "Flash Player";
  }
  return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Output: Flash Player 8.5
```

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 83.

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 reference

In many programming languages, the distinction between passing arguments by value or by reference is important to understand and 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 argument is made. Rather, 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 belonging to the primitive data types, which includes Boolean, Number, int, uint, and String, have special operators that make them behave as if they are passed by value. For example, the following code creates a function named `passPrimitives` that defines two parameters of type `int`. Two variables of type `int`, `xValue` and `yValue`, are sent as arguments to the `passPrimitives` function. Because they are primitives, they behave as if passed by value. Although the local parameters, `xParam` and `yParam`, initially contain only references to the `xValue` and `yValue` objects, any changes to the variables within the function body generates new copies of the values in memory.

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

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

Within the `passPrimitives` function, the values of the `xParam` and `yParam` variables 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 a parameter to the `passByRef` function. Because the object is not a primitive type, the object is passed by reference. This means that changes made to the parameters within the function will affect the object properties outside the function.

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

The `objParam` local variable references the same object as the global `objVar` variable. You can see from the `trace()` statements in the example that changes to the `x` and `y` properties of the `objParam` object are reflected in the `trace()` statement that executes in the global scope after the function returns.

Default parameter values

New for 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. If you call a function and take advantage of default values, your code compiles and runs, but does generate a compiler warning that the function expects a different number of arguments.

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

The arguments object

When parameters are passed to a function, you can use the arguments object to access information about the arguments passed to your function. There are three aspects of the arguments object:

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

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

```
function traceArgArray(x:int) {
  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 literal to enable recursion.

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

trace(factorial(5)); // Output: 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 arguments using the parameter names 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 such that an `arguments` parameter is added, then 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) {
```

```
  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 such that it passes an extra argument that is a reference to itself.

The ... (rest) parameter

ActionScript 3.0 introduces a new parameter declaration, the `...` (rest) parameter, that 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) {
  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 should skip 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 are 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 can also have properties and methods attached to it.

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

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

The `Array.sort()` method also uses a function as a parameter. 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 120.

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 called `length` that stores the number of parameters defined for the function. This is different from the `arguments.length` property, which reports the number of actual 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 and the number of parameters, shows the difference between the two properties:

```
function traceLength (x:uint, y:uint) {
```

```
  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 of 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, though 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 () {
  someFunction.counter++;
}

someFunction();
someFunction();
trace (someFunction.counter); // Output: 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 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 function definition, followed by the function in which it is defined. 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 static snapshot of a function and its *lexical environment*. A function's lexical environment includes all of the variables, properties, methods and objects in the function's scope chain along with their values. Though function closures are created any time a function is defined apart from an object or a class, they are most interesting when discussing nested functions. The fact that function closures retain the scope in which they were defined creates interesting results when a function is passed as an argument 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()` and returns the value `160` instead of `8`.

```
import flash.util.trace

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(); // Output: 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 this reference in a bound method always refers to the instance to which it was originally attached. For more information, see "Bound methods" on page 87.

CHAPTER 4

Object Oriented Programming in ActionScript

4

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Classes

Class definitions

Class attributes

Class body

Class property attributes

Access control namespace attributes

Prototype attribute

Static attribute

User defined namespace attributes

Variables

Static variables

Instance variables

Methods

Constructor methods

Static methods

Instance methods

Get and set accessor methods

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Bound methods

Interfaces

Defining an interface

Implementing an interface in a class

Inheritance

Instance properties are inherited

Overriding methods

Static properties are not inherited

The super statement

Advanced Topics

History of ActionScript OOP support

ActionScript 1

ActionScript 2.0

ActionScript 3.0

The class object

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

The prototype object

The prototype attribute

Prototype variables

Prototype methods

Prototypes and inheritance

Prototype properties compared to static properties

Sample: Class framework for samples in this book

CHAPTER 5

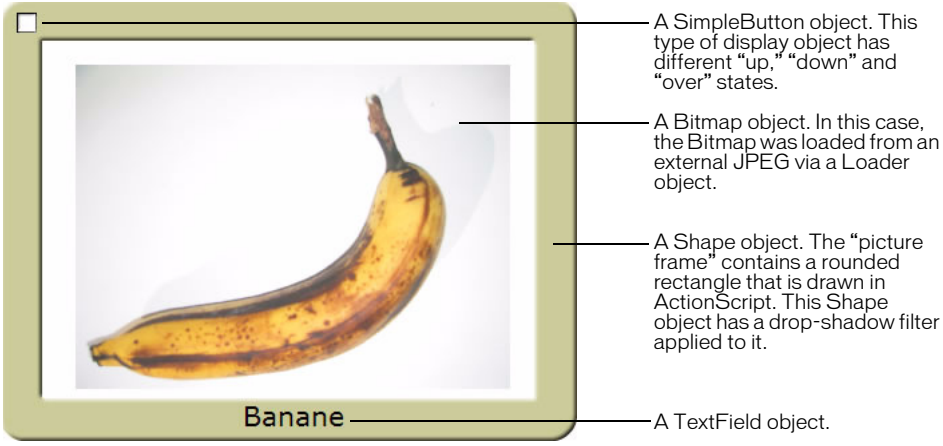
Display Programming

5

This chapter describes the basic concepts for working with onscreen elements in ActionScript 3.0. For information on manipulating onscreen elements visually, see "Modifying Display Objects" on page 197. For information on user interactivity with onscreen elements, see "Interactivity" on page 201.

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.

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 display object container object called a Sprite that contains the following display objects:

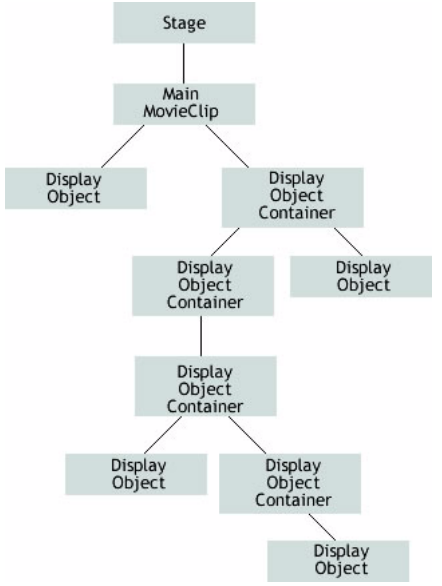


In the context of discussing display objects, DisplayObjectContainer objects are also known as *display object containers* or simply *containers*.

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.

Each application built with ActionScript 3.0 has a hierarchy of display objects, known as the *display list*.

At the top of the display list hierarchy is the Stage, which is the top-level container:



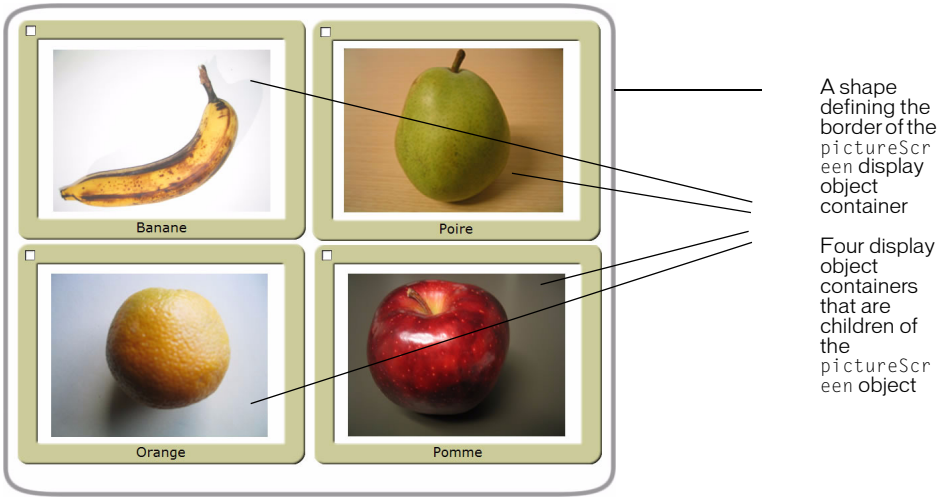
The main SWF file for an application (for example, the one that you embed in an HTML page) is automatically added as a child of the Stage. Typically it is a MovieClip, but it can be an object of any type that extends the Sprite class.

Any display objects that you create *without* using ActionScript, for example by adding a MXML tag in Flex Builder or by using a drawing tool in Flash, are added to the display list. You do not need to add them via ActionScript. However, you *can* access them via ActionScript. For example, the following adjusts the width of an object named `button1` that was added in the authoring tool (not via ActionScript):

```
button1.width = 200;
```

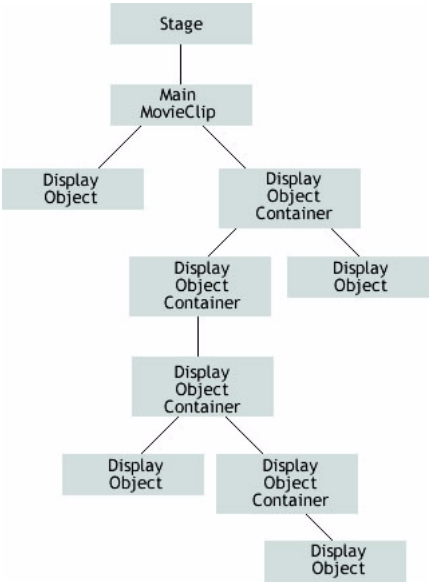
If a DisplayObjectContainer object is deleted from the Stage, or if it is moved or transformed in some other way, then each display object in the DisplayObjectContainer is also deleted, moved, or transformed.

A display object container is itself a type of display object—it can be added to another display object container. For example, the following shows a display object container, `pictureScreen`, that contains one outline shape and four other display object containers (of type PictureFrame):



Structure of the display list

The display list contains all visible display object in the application. It contains these objects in a hierarchical, tree structure:



The hierarchy reflects the front-to-back layering of display object. The Stage (at the top of the hierarchy) is in back of all other objects. For more information, see "Adding display objects to the display list" on page 96.

The Stage

Each application (the main SWF file loaded) has one Stage object, which is a display object container that contains all onscreen display objects. You can access the Stage via the `stage` property of any DisplayObject instance.

The Stage class overrides most properties and methods of the DisplayObject class to throw errors. This is because many of these properties do not apply to the Stage. 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 do not apply.

The `framerate` property of the Stage 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*.

Display object containers

Display object containers are special types of display object that can contain child objects that are also display objects. The DisplayObjectContainer class includes properties and methods for adding, deleting, and managing display objects in the display list of the display object container.

If a DisplayObjectContainer is deleted from the Stage, or if it is moved or transformed in some other way, then each display object in the DisplayObjectContainer is deleted, moved, or transformed along with it.

As mentioned in the previous section, the Stage is a display object container. Other classes that are subclasses of DisplayObjectContainer are Sprite, MovieClip, and Loader.

Display objects

The DisplayObject class includes properties and methods common to all display object (see "DisplayObject class" on page 102).

You can instantiate a display object, and it will not appear on screen (on the Stage) until you add the display object instance to a display object container that is a on the display list. For example, in the following code, the `myText` display object (a TextField object) is not visible if you were to omit the last line of code (in which the context of `this` must be a display object container that is already added to the display list):

```
import flash.display.*
var myText:TextField = new TextField();
myText.text = "Buenas dias.";
this.addChild(myText);
```

Advantages of the 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. Also, the ActionScript 3.0 display list allows you to create hierarchical structures of display objects, via container objects (which can contain other display objects and containers); whereas in ActionScript 2.0, visual objects within a movie clip are added in a "linear" stacking order (via depth levels).

This individualization of classes and hierarchical display list structure has the following benefits (described in the sections that follow):

- More efficient rendering and smaller file sizes
- Improved depth management
- Easier subclassing of display objects

More efficient rendering and smaller file sizes

Since a basic display object, such as a Shape object, does not include the full set of methods and properties that a MovieClip object has, it is less taxing on memory an processor resources. For instance, each MovieClip object includes properties for the timeline of the movie clip, whereas a Shape object does not. These properties for managing the timeline can use a lot of memory and processor resources. In ActionScript 2.0, you could only draw shapes in a MovieClip object. In ActionScript 3.0, using the Shape object, which has less overhead than the more complex MovieClip object, results in better performance: because Flash Player does not need to manage unused MovieClip properties, rendering speed improves and the file size reduces.

Improved depth management

In ActionScript 3.0, the depth of display objects is managed via the order of the child index numbers in display object containers. And each child can itself be a display object container.

In ActionScript 2.0, depth was managed via a linear depth management scheme and methods such as `getNextHighestDepth()`. There was no hierarchical arrangement of display objects (via containers), as there is in ActionScript 3.0.

In ActionScript 3.0 it is easy to change the depth of a group of related display objects at once, since they can be included in a display object container. When you move a container around in the display list (changing its parent container, for example) the children are moved with it.

Easier subclassing of display objects

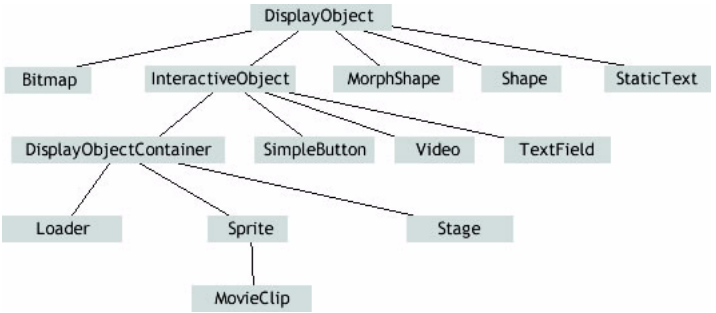
Note that in ActionScript 2.0, you would often have to add new MovieClip objects to a SWF file to create basic shapes or to display bitmaps. In ActionScript 3.0, the DisplayObject class includes many built-in subclasses, including Shape and Bitmap. Because the classes in ActionScript 3.0 are more specialized for specific types of objects, it is easier to create basic subclasses of the built-in classes.

For example, in ActionScript 2.0, you could create a CustomCircle class that extends MovieClip that could draw a circle 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:

```
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:Number) {
    xPos = xInput;
    yPos = yInput;
    radius = rInput;
    color = colorInput;
    this.graphics.beginFill(color);
    this.graphics.drawCircle(xPos, yPos, radius);
  }
}
```

Display Object Class Hierarchy

ActionScript 3.0 includes a number of subclasses of the base DisplayObject class:



For a description of these, see "Core display classes" on page 101.

Adding display objects to the display list

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, 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";
DisplayObjectContainer(this.root).addChild(myTextField);
```

In this code sample, `this.root` points to the MovieClip that contains the code (and a MovieClip is a type of display object container). 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 relate to the layering (the front-to-back order) of the display objects. For example, consider the following three display objects (each objects created from a custom class, Ball):

The layering of these display objects in their container can be adjusted using the `addChildAt()` method. For example, consider the following:

```
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 (note the layering of the objects):

You can use the `getChildAt()` method (which returns child objects of a container based on the index number you pass it) to verify the layer order of the display objects:

```
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, then the higher elements on the list each move down a position in the child index. For example, continuing with the previous code:

```
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 later in code. (Use the `delete` operator to completely remove an object.)

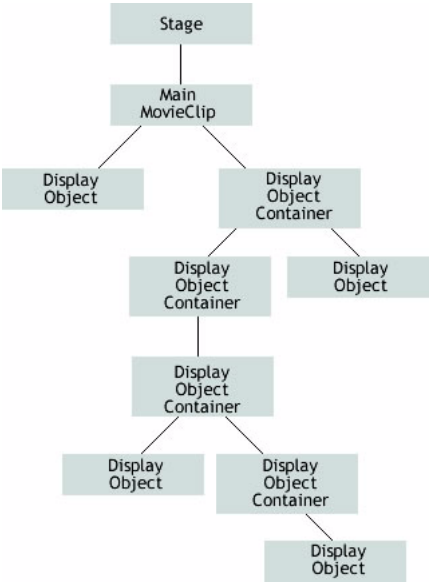
You can only add an instance of a display object to one display object container, since a display object has only one parent container. For example, note that the display object `tf1` can only exist in one container (Sprite extends DisplayObjectContainer):

```
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.

Traversing the Display List

The display list is a tree structure. At the root of the tree is the Stage, which can contain multiple display objects, and 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 through the display list, via the `child` lists of display object containers. For example consider the following:

```
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");
var pict.load(url);
var pict.name = "banana loader";
container.addChild(title);
container.addChild(pict);
```

This code adds two display objects (`title` and `pict`) to the `container` Sprite. 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"):

```
trace(container.getChildByName("banana loader") is Loader); // true
```

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 given, 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)`, and you can also write `container.getChildAt(0).getChildAt(0)` (since `container.getChildAt(0) == pict`).

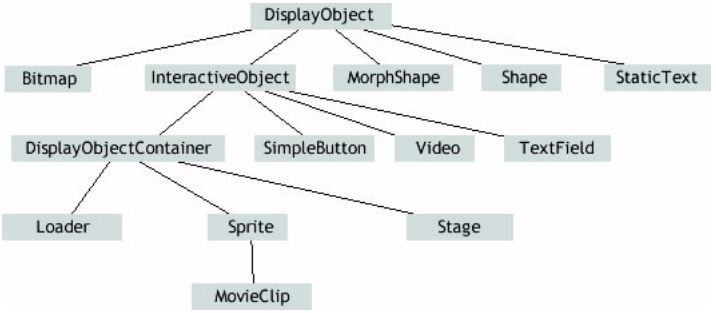
The following function provides an indented trace output of the display list from a given display object container:

```
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 + "")
    }
  }
}
```

Note for Flex users: 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 you are adding be a type of mx.core.UIComponent object.

Core display classes

The ActionScript 3.0 flash.display package includes classes for objects that can appear (visually) in the Flash Player window. The following shows the subclass relationships of various display object classes:



You can instantiate instances of the following classes contained in the flash.display package:

- Bitmap—Used to define bitmap objects, either loaded from external files or rendered via ActionScript. See "Bitmap class" on page 104.
- Shape—Used to vector graphics, such as rectangles, lines, circles, etc. See "The Shape class" on page 106.
- Loader—Used to load external assets (either SWF files or graphics). See "Loader class" on page 108.
- Sprite—A Sprite can contain graphics of its own, and it can contain child display objects. (Sprite is a type of DisplayObjectContainer). See "Sprite class" on page 103.
- MovieClip—A MovieClip has a timeline. Each SWF file loaded with the Loader class is a MovieClip instance, and so is the main SWF file loaded for the application. See "MovieClip class" on page 103.
- SimpleButton—A SimpleButton object has three button states: up, down, and over. See "SimpleButton class" on page 104.
- TextField—TextFields let you create display objects for text display and input. See "TextField class" on page 105.

Also, the Video class, which is included in the flash.media package, is a subclass of the DisplayObject class. See "Video class" on page 107.

This chapter includes basic information on these classes. For information on modifying display objects visually, see "Modifying Display Objects" on page 197.

Note that there are a other classes in the flash.display package, though you cannot instantiate them:

- DisplayObject—This is the base class for all display objects. For more information, see "DisplayObject class" on page 102.
- DisplayObjectContainer—The Sprite, Loader, and MovieClip class are all types of DisplayObjectContainers. For more information, see "Display object containers" on page 93.
- Graphics—The Graphics class includes methods for drawing vector shapes (for instance, to the `graphics` property of a Shape, Sprite, or MovieClip object). For more information, see "Graphics drawing methods" on page 198.
- StaticText—You cannot create a StaticText object directly in ActionScript. Static text fields are only created in the authoring tool. For more information, see "Working with StaticText objects" on page 198.
- InteractiveObject—The base class for all objects with which you can interact with the mouse and keyboard. SimpleButton, TextField, Video, Loader, Sprite, Stage, and MovieClip objects are all subclasses of InteractiveObject.
- MorphShape—These objects are created when you apply a shape tween. You cannot instantiate them via ActionScript.
- Stage—A type of DisplayObjectContainer. There is one Stage instance for an application, and it is at the top of the display list hierarchy. For more information, see "The Stage" on page 92.
- Other classes, such as TextFieldType, are used as support for the other classes in the flash.display package. For more information on these, see the *ActionScript 3.0 Language Reference*.

DisplayObject class

All display objects are subclasses of the DisplayObject class, and as such they inherit the properties and methods of the DisplayObject class. For details on many of the properties, see "Basic DisplayObject properties" on page 198.

These 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. Note, however, that the Stage class and the Loader class, which are both subclasses of the DisplayObject class, override many of these properties and methods to throw exceptions, as they do not apply to objects of those types. (For example, the Stage class does not have an `x` or a `y` property, since it is the top-level display object container—it is not a member of any other display object container for which these coordinates would apply.)

There is no constructor function for the DisplayObject class. You must create another type of object (that is a subclass of the DisplayObject type), such as a Sprite, to instantiate an object with the `new` constructor.

Sprite class

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.

You can draw shapes via the sprite's `graphics` property (for more information, see "Graphics drawing methods" on page 198). You can add other display object as children.

For example, the following Sprite has a circle drawn to 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 always appears in back of the child display objects of the Sprite. Also, the graphics layer does not appear in the child list of the Sprite. For more information on vector drawing capabilities of the Graphics class, see "Graphics drawing methods" on page 117.

A Sprite object is a type of display object container, which means that it is also a type of InteractiveObject. So it can respond to mouse, keyboard, and focus events. Additionally, a Sprite has a `buttonMode` property, which allows you to have the hand cursor appear when the mouse passes over the sprite. However, the SimpleButton class provides much more functionality in defining buttons, such as the ability to define separate up, down, and over states (see "SimpleButton class" on page 104). For more information on the interactivity capabilitys of the Sprite class, see "Interactivity" on page 201.

MovieClip class

The MovieClip class is a subclass of the Sprite class (see "Sprite class" on page 103). Each MovieClip object has a timeline, and a MovieClip object includes properties and methods for controlling the playhead and working with frames and scenes. Any SWF file you load is a MovieClip object.

Many properties and methods of the ActionScript 2.0 MovieClip class are present as properties or methods of the ActionScript 3.0 MovieClip class. Some are members of the MovieClip class because they are inherited by 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`, are renamed without the underscore in ActionScript 3.0. Also, the `_xmouse`, `_xscale`, `_ymouse`, and `_yscale` properties have been renamed `mouseX`, `scaleX`, `mouseY`, and `scaleY`.

You can set the frame rate for all movie clips in an application via the `frameRate` property of the Stage object.

Bitmap class

You can load bitmaps from external files, via the Loader class. You can load GIF, JPG, or PNG files. You can also create bitmaps, using methods of the BitmapData class. Whether loaded or created in ActionScript, you can use the methods of the BitmapData class to alter bitmaps.

For more information, see "Loader class" on page 108 and "Creating and manipulating bitmaps" on page 198.

SimpleButton class

In ActionScript 3.0, you can define button behavior via the SimpleButton class. A SimpleButton has three states: `upState`, `downState`, and `overState`. These are each properties of the SimpleButton object, and they are each DisplayObjects.

For example, the following class defines a simple text button:

```
private 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(flash.events.MouseEventType.MOUSE_UP,
                     changeTextButtonState);
  }

  public function changeTextButtonState(e:Event) {
    trace("Button clicked.");
  }
}
```

The `hitTestState` property of a SimpleButton is a DisplayObject that responds to the mouse events for the button. In this example, we set the `hitTestState` (and the `overState`) to be the same DisplayObject as the `upState`.

The TextButton class in the example references the following class, which defines the DisplayObject to be used for a state (up, down, or over) for the button:

```
private class TextButtonState extends Sprite {
  public var label:TextField;
  public function TextButtonState(color:uint, labelText:String) {
    label = new TextField();
    label.text = labelText;
    label.x = 3;
    var format:TextFormat = new TextFormat();
    format.font = "Verdana";
    label.setTextFormat(format);
    var buttonWidth:Number = label.getLineMetrics(0).width + 12;
    var background:Shape = new Shape();
    background.graphics.beginFill(color);
    background.graphics.lineStyle(1, 0x000000);
    background.graphics.drawRoundRect(0, 0, buttonWidth, 18, 2);
    background.filters = [new DropShadowFilter(1)];
    addChild(background);
    addChild(label);
  }
}
```

TextField class

The TextField class lets you work with dynamic and input text fields. For details, see "Text Fields" on page 198.

Note that there is also a StaticText class. However you cannot instantiate StaticText objects in ActionScript; these are created in the authoring tool.

The Shape class

A Shape object is a type of display object in which you can draw simple shapes: lines, fills, circles, rectangles, and so on. You can draw these via the authoring tool (such as Flash or Flex), or via ActionScript commands.

Each Shape object has a graphics property—an object that contains the lines and fills for the shape. The Sprite and MovieClip objects also have 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 draws an orange circle in a Shape and adds that shape to the Stage:

```
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.root.stage.addChild(circle);
```

For more information on the Graphics class and its drawing methods, see "Graphics drawing methods" on page 198.

Video class

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 class 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);
    stream.publish(false);

    var myVideo:Video = new Video(360, 240);
    myVideo.attachNetStream(stream);
    stream.play(videoUrl);

    addChild(myVideo);
  }
}
```

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 MovieClip or Bitmap object that it loads. By adding the Loader object to the display list, the child (SWF or Bitmap) object is also added 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 MovieClip or Bitmap is loaded, you can move the loaded object (the MovieClip or Bitmap instance) to another DisplayObjectContainer:

```
var pictLdr:Loader = new Loader();
var pictURL:String = "banana.jpg"
var pictURLReq:URLRequest = new URLRequest(pictURL);
pictLdr.load(pictURLReq);

pictLdr.addEventListener(flash.events.MouseEventType.CLICK, modifyPict);
function pictLoaded(e:Event):Void {
     addChild(pictLdr);
}
```

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:

- Via the `loadeeInfo` property of the Loader object
- Via the `loaderInfo` property of the loaded DisplayObject

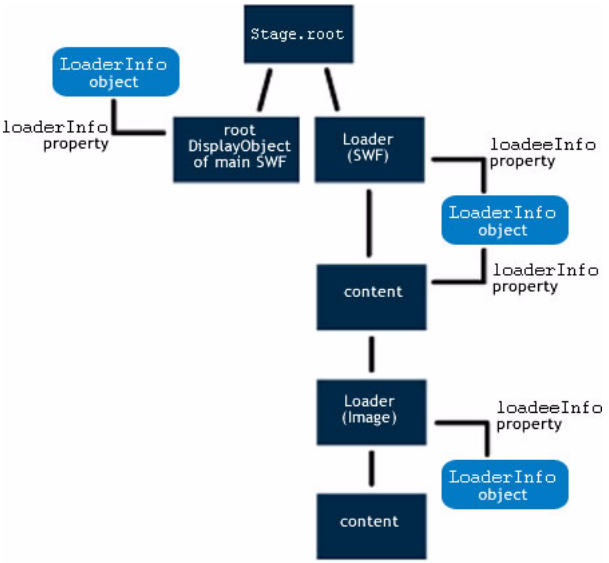
The LoaderInfo class provides information such as load progress, the URLs of the loader and loadee, the number of bytes total for the media, and the nominal height and width of the media.

When loading a SWF file by using a Loader object, the `loaderInfo` property of the loaded MovieClip object refers to the same LoaderInfo object as the `loadeeInfo` property of the Loader object. In other words, a LoaderInfo object is shared between a loaded SWF file and the SWF file that loaded it (between loader and loadee). The main (top-level) SWF file for an application has no Loader object associated with it.

For a Loader object that is used to load an image file rather than a SWF file, the DisplayObject created to hold the image has a null `loaderInfo` property, so the `loadeeInfo` property of the Loader object is the only way to access the LoaderInfo object for a loaded image.

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

This diagram shows the three different uses of the LoaderInfo object—for the main SWF file, for a SWF file Loader object, and for an image file Loader object:

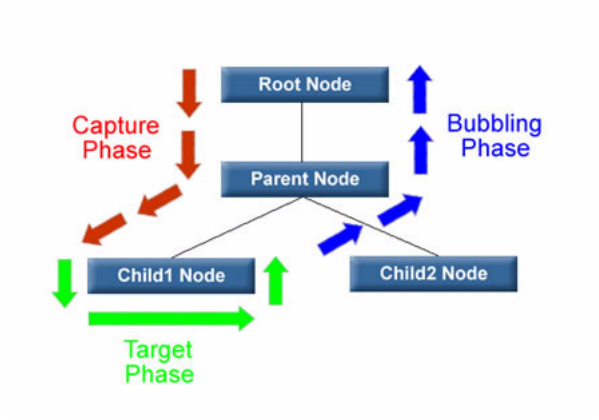


Display list API events and flow

The DisplayObject class inherits from the EventDispatcher class. This means that every display object can participate fully in the event model (described in "Event Handling" on page 211). 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 root of the display list 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 root node through the display list hierarchy down to the `child1` display object.

The event flow is conceptually divided into three phases:



For more information, see "Event Handling" on page 211.

Example: Creating custom display classes

More information to come in a future draft.

CHAPTER 6

Flash Player Security

6

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Overview of Permission Controls

Administrative User Controls

User Controls

Website Controls

Author (Developer) Controls

Security Sandboxes and Security Domains

Network SWF files

Local SWF files

Local-with-file-system

Local-with-networking

Local-trusted

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Permission mechanisms

SWF, image, sound, and video loading

Event Security

Data loading

Data sending

Shared objects

Cross-scripting APIs

Outbound scripting

Imported Runtime Shared Libraries

Camera, Microphone, and Clipboard Access

Options when publishing

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Security differences between Flash Player 8.5 and previous versions

PART 2

Core ActionScript Data Types and Classes

2

The following chapters are included:

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

CHAPTER 7

Core Classes Overview

7

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

Array class

Creating arrays

Inserting array elements

Removing array elements

Sorting an array

Querying an array

Date class

Date Constructor

Working with UTC dates

Date Class Static Methods

Date Class Instance Methods

Daylight Savings Time

Error classes

Math class

Namespace class

QName class

RegExp class

String class

XML class

XMLList class

CHAPTER 8

Math Functions

8

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

- "ActionScript Language and Syntax" on page 17
- "Display Programming" on page 89
- "Working with Strings" on page 127
- "Using Regular Expressions" on page 155
- "Working with XML" on page 179
- "Event Handling" on page 211
- "Networking and Communication" on page 227
- "Client System Environment" on page 249
- "Using the External API" on page 253

Math class

Example: Simple Calculator

CHAPTER 9

Working with Dates and Times

9

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

- "ActionScript Language and Syntax" on page 17
- "Display Programming" on page 89
- "Working with Strings" on page 127
- "Using Regular Expressions" on page 155
- "Working with XML" on page 179
- "Event Handling" on page 211
- "Networking and Communication" on page 227
- "Client System Environment" on page 249
- "Using the External API" on page 253

Date class

setInterval statement

Timer API

Example: Alarm Clock

CHAPTER 10

Working with Strings

10

The String class contains methods that let you work with text strings. Strings are important in working with many objects, and the methods described in this chapter are useful in working with strings used in many objects, such as TextField, StaticText, XML, Context Menu, and FileReference objects.

Flash Player 6 and later support Unicode characters. However, Flash Player 8.5 is the first version to support ActionScript 3.0.

This chapter includes the following sections:

Declaring strings

To declare a string literal, use straight single quotation mark (') or double 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:

```
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 (') and double quotation marks (") within a string literal, 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"';
```

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 characters to delineate string literals. When pasting text from another source into ActionScript, be sure to use the correct characters.

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
`\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

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 the characters in various positions in a string, 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:

```
var myStr:String =
  String.fromCharCode(104,101,108,108,111,32,119,111,114,108,100,33);
    // Sets myStr to "hello world!"
```

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);            // Output: 10
```

An empty string and a null string both have a length of zero:

```
var str1:String = new String();
trace (str1.length);           // Output: 0

str2:String = '';
trace (str2.length);           // Output: 0
```

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 shown in the following example:

```
var str1:String = "Apple";
var str2:String = "apple";
trace (str1 < str2) {
  trace("A < a, B < b, C < c, ...");
}
```

Use the == and != operators to compare strings with other types of objects:

```
var str:String = "4";
var total:uint = 4;
trace(str == total); // true
```

Use the === and !== operators to verify that both comparison objects are of the same type:

```
var str1:String = "4";
var str2:String = "4";
var total:uint = 4;
trace(str1 === str2); // true
trace(str1 === total); // false
```

Converting other objects to strings

You can convert any objects to a String representation. 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.

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:

```
str1:String = "green";
str2:String = "ish";
str3:String = str1 + str2;
    // str2 == "greenish"
```

You can also use the += operator to the produce the same result:

```
str1:String = "green";
str2:String += "ish";
    // str2 == "greenish"
```

Additionally, the String class includes a `concat()` method:

```
str1:String = "Bonjour";
str2:String = "from";
str3:String = "Paris";
str4:String = concat(str1, " ", str2, " ", str2)
    // str4 == "Bonjour from Paris"
```

If you use the + operator (or the += operator) with a string and an object that is *not* a string, ActionScript automatically converts nonstring object to a string 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 to force the + operator to evaluate arithmetically:

```
trace("Total: $" + 4.55 + 1.45);
    //"Total: $4.551.45"
trace("Total: $" + (4.55 + 1.45));
    //"Total: $6"
```

When you concatenate String objects, Flash Player allocates new memory for the new version of the string. With multiple concatenation operations, this allocation can result in unwanted memory usage by your Flash application. The StringBuilder class has better memory management techniques for working with text strings. For more information, see "The StringBuilder class" on page 138.

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 diameters):

```
/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 13, "Using Regular Expressions," on page 155.

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));
  // Output: 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));
  // Output: Pari
trace(str.slice(-3,-1));
  // Output: !!
    trace(str.slice(-3,26));
      // Output: !!!
    trace(str.slice(-3,str.length));
      // Output: !!!
```

```
trace(str.slice(-8,-3));
  // Output: Texas
```

Note that 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:

```
var str:String = "The moon, the stars, the sea, the land";
trace(str.indexOf("the"));
  // Output: 10
```

Note that `indexOf()` is case sensitive.

You can specify a second parameter to indicate the index position in the string from which to start the search:

```
var str:String = "The moon, the stars, the sea, the land"
trace(str.indexOf("the", 11));
  // Output: 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"));
  // Output: 30
trace(str.lastIndexOf("the", 29));
```

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));
  // Output: 21
```

Creating an array of substrings segmented by a delimiter

You can use the `split()` method to create an array of substrings of a string, which is divided based on a delimiter. For example, you can segment a comma- 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"]
```

Note that 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 13, "Using Regular Expressions," on page 155 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 13, "Using Regular Expressions," on page 155.

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:

```
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 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.

Note that 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, `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"]
```

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:

```
var str:String = "She sells seashells by the seashore.";
var pattern:RegExp = /sh/i;
trace(str.replace(pattern, "sch"));
  //sche sells seaschells by the seaschore.
```

You can include the following $ replacement codes in the replacement string. The replacement text shown is inserted in place of the $ replacement code.:

$ Code	Replacement Text
`$$`	`$`
`$&`	The matched substring.
`$``	The portion of the string that precedes the matched substring. Note that this code uses the straight left single quotation mark character (`), not the straight single quotation mark (') or the left curly single quotation mark (‘).
`$'`	The portion of the string that follows the matched substring. Note that 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.

You can also use a function as the second parameter of the `replace()` method, and 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));

private function usdToEuro(...args):String {
  var usd:String = args[1];
  usd = usd.replace(",", "");
  var exchangeRate:Number = 0.853690;
  var euro:Number = usd * exchangeRate;
  const euroSymbol:String = String.fromCharCode(8364);
  return euro.toFixed(2) + " " + euroSymbol;
}
```

Converting strings between uppercase and lowercase

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.
```

Note that after these methods are executed, the source string remains unchanged. To transform the source string, use the following code:

```
str = str.toUpperCase();
```

The StringBuilder class

When you modify String objects, for example by appending text, Flash Player allocates new memory each time the string is modified. This can result in unwanted memory usage by your Flash application.

The StringBuilder class is similar to the String class, but it has better memory allocation capabilities. Like a String object, a StringBuilder object holds text data. However, you can edit text in a StringBuilder object without increasing memory usage in the way that can happen when you edit text in a String object.

For example, each time you concatenate a String object, as shown in the following example, Flash Player allocates a new portion of memory for each new version of the String object, and the remaining versions of the String object are left in unused portions of memory:

```
var str:String = "Some string data. ";
for (var i:int = 0; i < 1000; i++) {
  var str += "Some more data. "
}
```

Flash Player may not immediately reallocate the unused portions of memory resulting from these String operations, because doing so could cause an undesired lag in performance.

Each StringBuilder object includes a capacity for a certain number of characters, tracked in the `capacity` property. This capacity may be larger than the current string in the StringBuilder object, and Flash Player adjusts the capacity to be even larger only as needed. For example, the StringBuilder class includes an `append()` method and an `insert()` method that let you append text to and insert text in a StringBuilder object. When you use these methods, the StringBuilder class allocates new memory only as needed, based on the existing capacity:

```
import flash.util.StringBuilder;
var sb:StringBuilder = new StringBuilder("Some string data. ");
for (var i:int=0; i<1000; i++) {
  sb.append("Some more data. ");
}
```

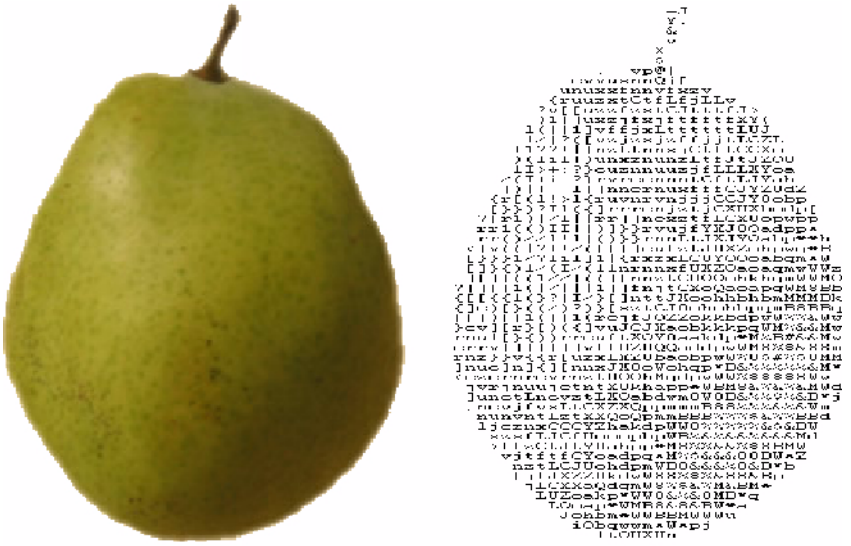
For details, see the section on the StringBuilder class in the *ActionScript 3.0 Language Reference*.

Sample: ASCII Art

This ASCII art sample shows a number of features of working with strings in ActionScript:

- A StringBuilder object is used for a large text string that is modified frequently.
- The methods of the String class are used to parse a string from a tab-delimited text file.
- The `getCharAt()` method is used to get a character from a string based on the grayscale value of a pixel of a bitmap.

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 ASCII art version of the graphic is shown on the right.

Running the sample file

The ASCII art sample file is prebuilt and ready to run.

To run the StringSample sample file:

1. Open the WorkingWithStrings folder, contained in the Chapters subfolder of the Prog_ActionScript folder.

   For more information on opening the sample files, see "Publishing and testing your ActionScript application" on page 16.
2. Open the project in your ActionScript 3.0 IDE.

3. Run the file.

   The sample displays a sequence of images. Along with each image, the sample displays the ASCII art version of the image and the image title.

   The images, along with some data for each image, are listed in a tab-delimited text file, "images.txt", located in the "txt" subfolder of the project.

Classes and assets used in the sample

The sample uses the following classes:

- StringSample.as and com.example.content.StringSampleContent.

  These classes are provide the basic structure for the sample, initializing the user interface, and setting up other objects.
- StringData and ASCIIData (both in the com.example.content package).

  These classes are used to load and parse the data in the "images.txt" file. These are described in the next section, "Loading and parsing the tab-delimited text file" on page 143.
- The ASCIIRecord class, which is used to normalize (process) the string data.

  This includes converting the extensions in the filename to lowercase, and capitalizing the titles. For more information, see "Using String methods to normalize the data" on page 144.
- The ASCIIPrinter class, which populates a StringBuilder object with ASCII art text, based on a bitmap image.

  This class processes data of the BitmapData property associated with any bitmap loaded into Flash Player. The class shows how to set the capacity of a StringBuilder object and how to add text to the object. For more information, see "Generating ASCII art text in a StringBuilder object" on page 146.
- Classes in the com.example.display package.

  This sample file uses the user interface class framework defined in the com.example.display package, common to all sample files in this book. For more information, see "Sample: Class framework for samples in this book" on page 88.

The sample uses the following external resources:

- The sample loads bitmap images from the "images" folder of the project.
- The "images.txt" file (in the "txt" folder of the project),

  The text file contains the following data:

```
FILENAMETITLEWHITE_THRESHHOLDBLACK_THRESHHOLD
fruit_basket.jpgPear, apple, ...d810
banana.jpgA picture of a bananaC820
orange.JPGorangeFF20
apple.jpgpicture of an apple6E10
```

  The file has a specific tab-delimited format. The first line (row) is simply a heading row. The remaining lines contain 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.

Building the sample

Although the sample provided is code complete, ready to run, this chapter describes how you can build the sample. Here are the high-level steps:

1. Implement the user interface framework, as defined in the com.example.display package.

   For more information, see "Sample: Class framework for samples in this book" on page 88.

2. Build the StringSample base class for the sample.

   The StringSample class extends the com.example.display.SampleFoundation class. Its constructor function instantiates the new StringSampleContent object:

```
protected override function init():void {
  title = "Chapter Ten: Working With Strings";
  description = "This is a description about Strings.";
  content = new StringSampleContent();
  super.init();
}
```

   The StringSampleContent class extends the SampleComponent class. Its init method overrides that of the SampleComponent class.

```
protected override function init():void {
  gutter = 5;
  imageStack = new ImageStack();
  var request:URLRequest = new URLRequest(dataTarget);
  data = new ASCIIData();
  data.addEventListener(Event.COMPLETE, completeHandler);
  data.load(request);

  printer = new ASCIIPrinter();
  addChild(printer);

  title = new SampleLabel();
  addChild(title);
}
```

   This `init()` method initialized the ASCIIData and ASCIIPrinter objects. These are described in the sections that follow.

3. Create code for loading the “images.txt” data file, and for parsing the data in this file.

   This is described in “Loading and parsing the tab-delimited text file” on page 143

4. Create code for normalizing the text data.

   This is described in “Using String methods to normalize the data” on page 144

5. Create code for generating the ASCII art versions of the images.

   This is described in “Generating ASCII art text in a StringBuilder object” on page 146

Loading and parsing the tab-delimited text file

Before it can display the images and the corresponding titles and ASCII art versions, the sample must load and parse the text in the "images.txt" data file.

To load and parse the tab-delimited text file:

1. In the StringSampleContent sample, create an ASCIIData object. Pass a URLRequest with the relative address of the text file ("txt/ImageData.txt") to the ASCIIData constructor function.

   The ASCIIData object loads the text file into a String instance named `data`.

2. Create an array named `rows` containing text from each line of the `data` string.

   The sample includes a `parse()` method, which assigns data to the array by implementing the `split()` method of the `data` string:

   ```
   var rows:Array = data.split("\n");
   ```

   Each element in the array is a string containing a line of text from the data string. (The `"\n"` newline character defines the boundaries of lines.)

3. Create a `records` array, in which each element of the array is an object in which each object corresponds to one data item for the image.

   The `parseRecord()` method accomplishes this:

   ```
   var properties:Array = row.split("\t");
   var record:ASCIIRecord = new ASCIIRecord();
   var len:uint = properties.length;
   for(var i:uint; i < len; i++) {
     record[accessors[i]](properties[i]);
   }
   ```

   Again, the code uses the `split()` method, this time to get an array of properties, delineated in each line by the tab, `"\t"`, character.

Using String methods to normalize the data

The ASCIIRecord class defines the object containing data for each record. Each object contains the following properties, which correspond to the rows of the input data: `fileName`, `title`, `whiteThreshold`, and `blackThreshold`.

To normalize string data by adjusting capitalization:

1. For each filename, convert the file extension portion of the filename to lowercase.

   In the ASCIIRecord class, the `setFileName()` method performs this conversion:

   ```
   public function setFileName(str:String):void {
     var fileHalves:Array = str.split(".");
     fileName = fileHalves[0] + "." + fileHalves[1].toLowerCase();
   }
   ```

2. Adjust the capitalization of the strings in the title properties of each record.

   In the ASCIIRecord class, the `normalizeTitle()` method accomplishes this. First it creates an array of words (by splitting on the space character),

   ```
   var words:Array = str.split(" ");
   var len:uint = words.length;
   ```

   Then it capitalized the first letter of each word:

   ```
   for(var i:uint; i < len; i++) {
     words[i] = capitalizeFirstLetter(words[i]);
   }
   ```

   The `capitalizeFirstLetter()` method capitalizes the first letter of the word, by using two methods of the String class—`toUpperCase()` and `substr()`:

   ```
   word.charAt(0).toUpperCase() + word.substr(1)
   ```

   However, it only does so if the word is not a special case that should not be capitalized:

   ```
   if(!isLowerCase(word)) {
     word = word.charAt(0).toUpperCase() + word.substr(1);
   }
   ```

   In English, the initial character of each word in a title is *not* capitalized if it is one of the following words: “of,” “at,” in,” “and,” “or,” “a,” or “an.” (This is a simplified version of the

rules.) The `isLowerCase()` method of the ASCIIRecord class determines whether a word (passed to it as a parameter) is in the exception list:

```
private static function isLowerCase(word:String):Boolean {
  if(lowerCaseWords == null) {
    lowerCaseWords = new Object();
    lowerCaseWords["and"] = true;
    lowerCaseWords["the"] = true;
    lowerCaseWords["in"] = true;
    lowerCaseWords["an"] = true;
    lowerCaseWords["or"] = true;
    lowerCaseWords["at"] = true;
    lowerCaseWords["of"] = true;
    lowerCaseWords["a"] = true;
  }

  return (lowerCaseWords[word] == true);
}
```

Generating ASCII art text in a StringBuilder object

Now that the data is normalized, the sample can display the image, its title, and the ASCII art version, one image at a time.

To generate a StringBuilder object with the ASCII art string:

1. Get the grayscale value of each pixel.

   The pixel-to-grayscale algorithm used by the `parseBitmapData()` method does not use any real String functionality. However, the algorithm produces an index grayscale value (`grayVal`) that is an unsigned integer (uint) between 0 and 255 (hexadecimal 0xFF).

2. This method produces a string, stored in the `asciiArtStr` StringBuilder object, that graphically approximates the BitmapData object passed as the parameter of the function.

   The pixel-to-grayscale algorithm used by this method does not use any real String functionality. However, the algorithm produces an index grayscale value that is an unsigned integer (uint) between 0 and 255 (hexadecimal 0xFF).

   For more information on BitmapData objects, see the *ActionScript 3.0 Language Reference.*

3. Based on the grayscale value of a pixel, select the appropriate ASCII character.

   The `replacementChars` string lists 64 characters that increase in darkness from a space character (which is totally white) to the `@` character (which is the "darkest" standard ASCII character):

   ```
   var replacementChars:String =
       " .,;-~_+<>i!lI?/
     \|()1{}[]rcvunxzjftLCJUYXZOOQoahkbdpqwm*WMB8&%$#@";
   ```

   These lines, in the `parseBitmapData()` method, appends the appropriate character from the `replacementChars` string based on the grayscale value stored in the index variable:

   ```
   index = uint(replacementChars.length - (grayVal / 4));
   print(replacementChars.charAt(index));
   ```

   They call the `print()` method of the ASCIIPrinter class:

   ```
   private function print(str:String = ""):void {
     buffer.append(str);
   }
   ```

   Note that the `buffer` object is a StringBuilder object. This means that you can replace its contents for each bitmap loaded, and the StringBuilder class does not use additional memory each time. Before any text is added to the buffer object, the code sets the capacity of the object, based on the size of the bitmap image and the resolution of the ASCII art representation:

   ```
   var capacity:uint = (data.height / verticalResolution)
                       * (1 + Math.ceil(data.width / horizontalResolution);
   buffer.ensureCapacity(cap);
   ```

4. At the end of each line of the ASCII Art (contained in the buffer object), add a newline character.

   The `println()` method does this, by using the `append()` method of the StringBuilder class:

   ```
   private function println(line:String = ""):void {
     buffer.append(line + "\n");
   }
   ```

   For more information on the StringBuilder class, see "The StringBuilder class" on page 138.

CHAPTER 11

Working with Arrays

11

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

- "ActionScript Language and Syntax" on page 17
- "Display Programming" on page 89
- "Working with Strings" on page 127
- "Using Regular Expressions" on page 155
- "Working with XML" on page 179
- "Event Handling" on page 211
- "Networking and Communication" on page 227
- "Client System Environment" on page 249
- "Using the External API" on page 253

Array class

Array functions

Example: Manipulating Arrays

CHAPTER 12

Handling Errors

12

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

Types of errors

Advantages of error handling

Synchronous errors vs. error notifications

The throw statement

try...catch...finally statements

The throw statement

Uncaught exceptions

Error classes

ECMAscript core error classes

ReferenceError class

EvalError class

ActionScript core error classes

StackTraceElement class

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

flash.error package error classes

Debugger error display differs from release error display

Creating your own specialized Error classes

Displaying errors to the user

Error handling strategies

Example: Function Test Harness

CHAPTER 13

Using Regular Expressions

13

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 many 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 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 134.

This chapter includes the following sections:

Introduction to regular expressions

A regular expression describes a pattern of characters. 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 called 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 165.

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 171.

You can use regular expressions to search for patterns in strings, and to replace characters, by using 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)); // 7

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 134.

The RegExp class includes the following methods: `test()` and `exec()`. For more information, see "Methods for using regular expressions with strings" on page 175.

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 two regular expressions are equivalent:

```
var pattern1:RegExp = /bob/i
var pattern2:RegExp = new RegExp("bob", "i");
```

Forward slashes delineate a regular expression literal, similarly to the way that 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:

```
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 characters 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 escape quotation marks in regular expressions that are defined by using the forward slash delineators. Similarly, do not escape forward slashes within 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.
```

The sections that follow describe syntax for defining regular expression patterns.

For more information on flags, see "Flags and properties" on page 171.

Characters, metacharacters, and metasequences

The simplest regular expression is one that matches a sequence of characters:

```
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 these metacharacters without their 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/
```

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

A *metasequence,* like a metacharacter, has special meaning in a regular expression, but a metasequence is made up of more than one character. The following sections provide details on metacharacter and metasequence use.

Overview of 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 172). Note that when used at the start of a character class, the caret indicates negation, not the start of a string (see "Character classes" on page 163).
$ (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 (see "The m (multiline) flag" on page 172).
\ (backslash)	Escapes the special metacharacter meaning of special characters.
. (dot)	Matches any single character. A dot matches a newline character (`\n`) only if the `s` (dotall) flag is set (see "The s (dotall) flag" on page 173).
* (star)	Matches the previous item repeated zero or more times. For more information, see "Quantifiers" on page 165.
+ (plus)	Matches the previous item repeated one or more times. For more information, see "Quantifiers" on page 165.
? (question mark)	Matches the previous item repeated zero or one time. For more information, see "Quantifiers" on page 165.
(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 167.

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 163.
`|` *(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`.

Metasequences

Metasequences are sequences of characters that have special meaning in a regular expression pattern, like metacharacters (which consist of one character each). The following table describes these metasequences:

Metasequence	Description
`{n}` `{n,}` *and* `{n,n}`	Used to identify a specific 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 165.
`\A`	Matches at the start of the string to which the regular expression is applied.
`\b`	Matches at the position between a word character and a non-word 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 non-word characters.
`\d`	Matches a digit.
`\D`	Matches any character other than a digit.
`\n`	Matches the newline character.
`\r`	Matches the return character.
`\s`	Matches any whitespace character (a space, tab, newline, or return character).
`\S`	Matches any character other than a whitespace 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.
`\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*.

Metasequence	Description
`\Z`	Matches the end of the string to which the regular expression is applied. If the string ends with a line break, matches *before* the final line break.
`\z`	Matches the end of the string to which the regular expression is applied. If the string ends with a line break, matches *after* the final line break.

Character classes

You use character classes to specify a list of characters to match one position in the regular expression. You define them with square brackets (`[` and `]`). For example, the following regular expression 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 meaning inside a character class. For example, the following character class matches the asterisk (which is used for repetition in a regular expression) literally, along with any of the other characters listed:

```
/[abc*123]/
```

However, these three characters *are* metacharacters (with special meaning) in character classes:

Metacharacter	Meaning
`]`	Defines the end of the character class.
`-`	Defines a range of characters (see "Ranges of characters in character classes" on page 164).
`\`	Used to define metasequences and to escape 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 `-`):

```
/[$\\\]\-]/
```

Also, the following metasequences *do* maintain special meanings within character classes:

Metasequence	Character
`\n`	Newline
`\r`	Return
`\t`	Tab
`\u`*nnnn*	The character with the specified Unicode code point value (as defined by the hexidecimal number *nnnn*)
`\x`*nn*	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 `a-z` or any digit:

```
/[a-z0-9]/
```

You can also use the `\x`*nn* 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, it simply adds 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:

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 or one time.
{*n*} {*n*,} and {*n*,*n*}	Used to identify a specific 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:

```
<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 173.

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 following 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 167.

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 163.

Groups

You can specify a group in a regular expression by using parentheses:

```
/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 each of 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:

```
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 abababab.

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 165.

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:

```
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 non-capturing groups and lookahead groups

A non-capturing group is one that is used for grouping only; it is not "collected," and it does not match numbered backreferences. Use `(?:` and `)` to define non-capturing groups:

```
var pattern = /(?:com|org|net);
```

For example, note the difference between putting `(com|org)` in a capturing versus a non-capturing 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

//Non-capturing:
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 non-capturing 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);
flash.util.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);
flash.util.trace(result.name); // bob
flash.util.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

There are 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:

```
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.

```
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 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 words 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:

- Carriage-return (`\r`)
- Unicode line-separator (`\u2028`)
- Unicode paragraph-separator (`\u2029`)

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:

```
/<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, escape the blank space with a backslash. For example, the following two regular expressions are equivalent:

```
/foo bar/
/foo \ bar/x
```

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` (`global`) 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, so the output for this 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:

```
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:

```
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:
// 5  8  She
// 15  24  seashells
// 32  40  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 134.

Example: Form validation using regular expressions

More information will be included in an upcoming draft of this chapter.

CHAPTER 14

Working with XML

14

With ActionScript 3.0, there is a new group of classes based on the ECMAScript Edition 4 draft language specification. These classes include powerful and easy-to-use functionality for working with XML data. Using E4X, you may find that you are able to develop code work with XML data faster than was possible with previous programming techniques, and that the code is easier to read.

This chapter includes the following sections:

- E4X: A new approach to XML processing
- XML objects
- XMLList objects
- Initializing XML variables
- Assembling and transforming XML objects
- Traversing XML structures
- XML type conversion
- Using XML namespaces
- Example: Loading RSS data from the internet
- Example: Using XML to access ActionScript class information

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 *very* 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`, and the `author` element has two child elements. The `book` element has an `ISBN` *attribute* (with values `"1"`, `"2"`, and `"3"`).The *content* of the `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://ww.example.com/xml", and this namespace defines the schema for this type of XML document.

ActionScript 3.0 includes new operators (such as `.` and `@` 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:

```
import flash.util.trace;
trace(order1.book[0].author.firstName); // Chuck
trace(order1.book.(@ISBN=="0865436401").pageCount; // 115
delete order1.book[0];
```

These 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 (such as a Flash Player and an application on a web server) via a common standard such as SOAP, the Simple Object Access Protocol.

E4X: A new approach to XML processing

Until now, the ECMAScript language specification (also known as ECMA-262), which is the basis for the core ActionScript 3.0 language, had no way of working with XML data. Prior versions of ActionScript (from ActionScript 1.0 in Flash 5 onward) had classes and methods for working with XML data, but they were not based on the ECMAScript standard.

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

The new ECMAScript Edition 4 draft language specification defines a new set of classes and functionality for working with XML data. These classes and functionality are known collectively as *E4X* ("ECMAScript for XML"). ActionScript 3.0 includes the new E4X classes: XML, XMLList, QName, and Namespace.

The methods, properties, and operators of the E4X classes were developed 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 is internally consistent and consistent with other parts of ActionScript.
- Familiarity—You manipulate XML data with well-known operators, such as the dot (.).

NOTE

There was an XML class in ActionScript 2.0, and in ActionScript 3.0 it has been renamed *XMLDocument*, so that it does not conflict with the new XML class that is part of E4X. In ActionScript 3.0 the legacy classes–XML, 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 *ActionScript 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 an RSS feed. However, the examples in this chapter assign XML data as literals, for clarity.

E4X includes some intuitive operators, such as `.` and `@` (known 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:

```
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:

```
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:

```
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>
```

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 whitespace characters are ignored in elements tags and embedded expressions that are separated only by whitespace 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: `appendChild()`, `child()`, `childIndex()`, `children()`, `descendants()`, `elements()`, `insertChildAfter()`, `insertChildBefore()`, `parent()`, and `prependChild()`.

The following methods allow you to work with attributes: `attribute()`, `attributes()`.

The following methods allow you to you work with properties: `hasOwnProperty()`, `propertyIsEnumerable()`, `replace()`, and `setChildren()`.

The following methods are for working with qualified names and namespaces: `addNamespace()`, `inScopeNamespaces()`, `localName()`, `name()`, `namespace()`, `namespaceDeclarations()`, `removeNamespace()`, `setLocalName()`, `setName()`, and `setNamespace()`.

The following methods are for working with and determining certain types of XML content: `comments()`, `hasComplexContent()`, `hasSimpleContent()`, `nodeKind()`, `processingInstructions()`, and `text()`.

The following are for conversion to strings and for formatting the XML object: `defaultSettings()`, `setSettings()`, `settings()`, `normalize()`, `toString()`, and `toXMLString()`.

There are a few additional methods: `contains()`, `copy()`, `valueOf()`, and `length()`.

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 properties and methods of the XMLList class are the same as those of the XML class.

An XML object is much like an XMLList object that contains only one item.

Initializing XML variables

You can assign an XML literal to an XML object:

```
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>
```

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, there is a missing a closing tag), you will see a run-time error.

You can also pass data by reference (from other variables) into an XML object:

```
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:

```
var externalXML:XML;
var loader:URLLoader = new URLLoader();
var request:URLRequest = new URLRequest("xmlFile.xml");
loader.load(request);
loader.addEventListener(EventType.COMPLETE, onComplete);

function onComplete(event:Event):Void {
    var loader:URLLoader = URLLoader(event.target);
    externalXML = new XML(loader.data);
    trace(externalXML.toXMLString());
}
```

To read XML data, 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:

```
var x1:XML = <p>Paragraph 1</p>
var x2:XML = <p>Paragraph 2</p>
```

```
var x:XML = <body></body>
x = x.appendChild(x1);
x = x.appendChild(x2);
x = x.prependChild(<p>Paragraph 0</p>);
  // x == <body><p>Paragraph 0</p><p>Paragraph 1</p><p>Paragraph 2</p></
  body>
```

Use the `insertChildBefore()` method or the `insertChildAfter()` method to add a property before or after a specified property:

```
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>)
```

You can also use curly brackets (`{` 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:

```
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 x: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 `x.book` is an XMLList containing child properties of the x object that have the name `book`, which are two XML objects (the two `book` properties).

The object `x..lastName` is an XMLList containing any descendent properties with the name `lastName`, which are two XML objects (the two `lastName` properties).

The object `x.book.editor.lastName` is an XMLList containing any children with the name `lastName` of children with the name `editor` of children with the name `book` of the `x` object: in this case, an XMLList 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 `x` that has two child properties named `book`. Each child property named `book` has an index number associated with it:

```
x.book[0]
x.book[1]
```

To access a specific grandchild you can specify index numbers for both the child and grandchild names:

```
x.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:

```
x.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:

```
x.book.title
```

You can use the `child()` method and to navigate to children with names based on a variable or expression:

```
var x.XML =
     <order>
       <book>
         <title>Dictionary</title>
       </book>
     </order>

var childName:String = "book";

trace (x.child(childName).title) // Output: Dictionary
```

Filtering by attribute or element value

You can use the `(` and `)` operators 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 are all valid:

```
x.employee.(lastName == "McGee") // The first employee node
x.employee.(lastName == "McGee").firstName // The firstName property of
  that node
x.employee.(lastName == "McGee").@id // The value of the id attribute
x.employee.(@id == 347)
x.employee.(@id == 347).lastName
x.employee.(@id > 300) // An XMLList with both employee properties
x.employee.(position.toString().search("analyst") > -1) // An XMLList with
  both position properties
```

Using the for ... in statement and the for each ... in statement

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 `myXML.item` (which is the XMLList object consisting 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* 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 the with statement

Use the `with` statement with a given XML object to evaluate properties and attributes you specify based that object. For example, consider the following 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>;
```

Use the `with` statement to reduce code. Consider this code, which does not use the `with` statement):

```
var total:Number = 0;
var outStr:String;
for each (var property:XML in myXML.item) {
  total = property.@quantity * property.price;
  outStr = property.@quantity + " " + property.menuName
         + " (ID:" + property.@id + ") = "
         + total.toFixed(2);
  trace(outStr);
}
```

This code is equivalent, but by using the `with` statement, you need not repeat "property." in the code:

```
var total:Number = 0;
var outStr:String;
for each (var property:XML in myXML.item) {
```

```
  with (property) {
    total = @quantity * price;
    outStr = @quantity + " " + menuName
           + " (ID:" + @id + ") = "
           + total.toFixed(2);
  }
  trace(outStr);
}
```

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:

```
// Create a SOAP message
var message:XML =
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <soap:Body>
    <w:GetTemparature xmlns:w="http://www.test.com/weather/">
      <city>SFO</city>
    </w:GetTemparature>
  </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 weatherNS:Namespace = new Namespace ("w",
                                  "http://www.test.com/weather/");
message.addNamespace(weatherNS);
var encodingStyle:XMLList = message.@soapNS::encodingStyle;
var body:XMLList = message.soapNS::Body;

// Change the city:
message.soapNS::Body.weatherNS::GetTemparature.city = "LAX";
```

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 Strings. 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. Each of these topics is discussed in the following sections.

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()` does exactly the same as `toXMLString()`. For XML objects with simple content (those that contain only one text element), the `toString()`method returns only the text content of the element:

```
var myXML:XML =
  <order>
    <item id='1' quantity='2'>
      <menuName>burger</menuName>
      <price>3.95</price>
    </item>
  <order>

trace(myXML.item[0].menuName.toXMLString())
  // Output: <menuName>burger</menuName>
trace(myXML.item[0].menuName.toString())
  // Output: burger
```

Converting strings to XML objects

You can use the `new()` constructor to create an XML object from a string:

```
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:

```
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 strings, and you may need to convert these to other data types. For example, the following code converts the text values to numbers (by using the `Number()` function):

```
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, it 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 and writing external XML documents

You can use the URLLoader class to load XML data from a URL (replace the `XML_URL` value in this 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) {
    myXML = XML(myLoader.data);
    trace("Data loaded.");
}
```

Example: Loading RSS data from the internet

More information will be included in an upcoming draft of this chapter.

Example: Using XML to access ActionScript class information

More information will be included in an upcoming draft of this chapter.

PART 3

Flash Player APIs

3

The following chapters are included:

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

CHAPTER 15

Modifying Display Objects

15

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

This chapter describes many of the properties and methods that you can use to visually manipulate display objects.

For basic information on display programming see "Display Programming" on page 89. For information on user interactivity with onscreen elements, see "Interactivity" on page 201.

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Basic DisplayObject properties

Bitmaps

Creating and manipulating bitmaps

Buttons

Text

Text Fields

Supported HTML tags

Supported HTML entities and character codes

Working with StaticText objects

Shapes

Graphics drawing methods

Sprites

Movie Clips

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Videos

Example

CHAPTER 16

Interactivity

16

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

This chapter discusses the ways in which the user can interact with display objects. Users can interact with objects in the InteractiveObject class via the mouse and keyboard.

For basic information on display programming see "Display Programming" on page 89. For information on modifying onscreen elements, see "Modifying Display Objects" on page 197.

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Focus and tab order

Mouse events

Keyboard events

Hypertext in HTML Text Fields

Example

CHAPTER 17

Flash Player API Overview

17

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

flash.accessibility package

flash.display package

flash.events package

flash.filters package

flash.geom package

flash.media package

flash.net package

flash.print package

flash.swf package

flash.system package

flash.text package

flash.ui package

flash.util package

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

flash.xml package

CHAPTER 18

Working with Geometry

18

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

Using Point objects

Finding the distance between two points

Translating coordinate spaces

Moving a display object by a specified angle and distance

Other uses of the Point class

Using Rectangle objects

Resizing and repositioning Rectangle objects

Finding unions and intersections of Rectangle objects

Other uses of Rectangle objects

Using Matrix objects

Defining Matrix objects

Defining a Matrix for use with a gradient

Example: Creating a gradient fill and flipping a display object around an axis

CHAPTER 19

Event Handling

19

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 Flash Player 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 into five sections. The first two, the introduction and the history section, 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 the reader has a basic understanding of the display list.

Introduction

There's a good chance that the next SWF file you create will 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. ActionScript 3.0 uses a system of events and event listeners that enables you to respond to user input. Events can be thought of as occurrences of any kind in your SWF file that are of interest to you as a programmer. For example, any user interaction with your SWF file will generate 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. You can “listen” for these events in your code using event listeners. Event listeners are the functions or methods that you write to respond to specific events.

Previous versions of ActionScript provided several different methods for handling events. Some could only be used in specific ways, and others could only be used with a handful of classes. Moreover, some methods had idiosyncracies that felt counterintuitive to many programmers. The previous event handling systems did not provide a cohesive event model that could be applied uniformly throughout the language.

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.

There are three basic ideas that form the structure of the new event model: the event flow, event objects, and event listeners. The event flow describes how the event notification system works within the structure of the display list. Event objects are the event model's basic units that Flash Player creates and dispatches into the event flow whenever an event occurs. Event listeners are what you create to listen for, and respond to, events.

Although the new event model allows advanced programmers to create and dispatch custom events, this chapter focuses exclusively on the events that are defined by ActionScript 3.0 and dispatched directly by Flash Player.

A brief history of event handling in ActionScript

Previous versions of ActionScript provided a number of different ways to handle events:

- on() event handlers that can be placed directly on button and movie clip instances;
- onClipEvent() handlers that can be placed directly on movie clip instances;
- callback function properties such as XML.onload and Camera.onActivity;
- event listeners registered using the `addlistener()` method; and
- the UIEventDispatcher class that partially implemented the DOM event model.

Each of these methods 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 onto buttons and movie clips can be difficult to find. Callback functions are also simple to implement, but limit developers to only one callback function for any given event. Event listeners are more difficult to implement—requiring 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 developers to create several listener objects and register them all for the same event.

The development of the 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 often overlapped in some ways, but differed 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. Yet, the syntax for registering listener objects differed depending on whether the developer was using one of the six classes that supported 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 named `addEventListener()`.

Another complexity of having several different event handling models was that the scope of the event handler function varied widely depending on the method used. In other words, the meaning of the `this` keyword varied widely depending on the event handling system you used.

The ActionScript 3.0 event model simplifies the event handling system by replacing the entire set of ActionScript 2.0 event handlers with a single event handling mechanism based on the DOM Level 3 Events specification.

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 directly to a URLStream object. If the event target is on the display list, however, Flash Player dispatches the event into the display list and the event travels through the display list to the event target. The event flow describes how that event object moves through the display list. The display list is organized in a hierarchy that can be described as a tree. Almost all users of computers today are familiar with this hierarchical system, as it is the basis for all the major desktop computer file systems.

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. Each Flash application can have only one stage.

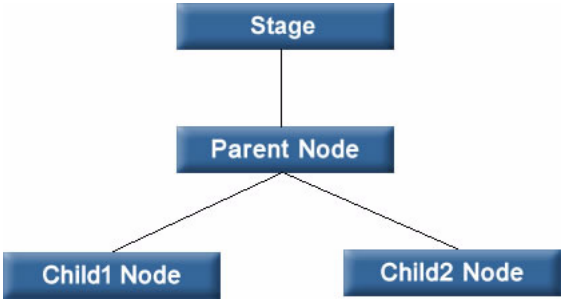
The DOM Events specification uses the term *node* to mean an item in a tree structure that can listen for events. Although, the term node isn't officially a part of ActionScript 3.0, it is used in the following discussion to highlight the similarities between the DOM Events specification and the ActionScript 3.0 event model. A node is an object on the display list.

Every object on the display list can trace its class inheritance back to the DisplayObject class. The DisplayObject class, in turn, inherits from the EventDispatcher class. The significance of this is that every item on the display list can participate fully in the event model. Every object on the display list can use its `addEventListener()` method—inherited from the EventDispatcher class—to listen for a particular event, but only if the listening object is part of the event flow for that event.

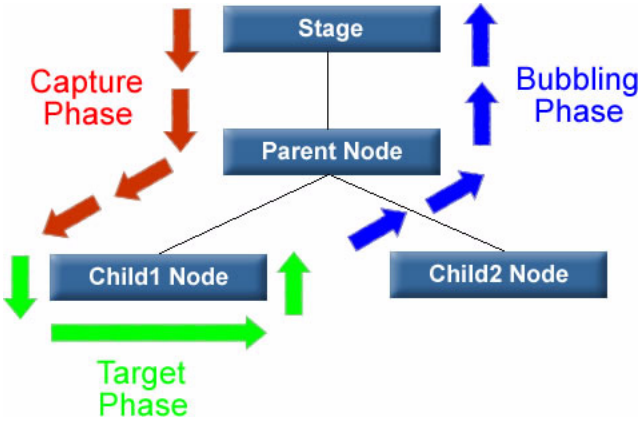
When Flash Player dispatches an event object, that event object makes a round-trip journey from the stage to the target node. The target node is the the display list object where the event occurred. For example, if a user clicks on a display list object named child1, Flash Player will dispatch an event object using child1 as the target node.

The event flow is conceptually divided into three parts. The first part is called the capture phase, which 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, which comprises all of the nodes encountered on the return trip from the node after the target node to the root node.

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. The object's journey starts at `Root Node`, moves down to `Parent Node`, then moves to `Child1 Node`, 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 diagram above shows the capture phase in red). The target phase comprises the time spent at `Child1 Node` (the target phase is green in the diagram above). The bubbling phase comprises `Parent Node` and `Stage` as they are encountered during the upward journey back to the root node (the bubbling phase is blue in the diagram above).

The event flow contributes to a more powerful event handling system than that previously available to ActionScript programmers. Prior to ActionScript 3.0, event listeners could only be registered on the object that generated a particular event. In ActionScript 3.0, not only can you register event listeners on a target node, but you can also register event listeners on any node along the event flow.

Not all event objects, however, participate 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 nor the bubbling phases. 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 or bubbling phases.

You can either reference the API documentation to find out how a particular event type behaves, or you can examine the event object's properties, as described in the following section.

The event object

The Event class 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.

```
package flash.events
{
  public class Event extends EventRoot
  {
    // constructor
    Event(type:String, bubbles:Boolean=false, cancelable:Boolean=false);

    // class constants
    public static const ACTIVATE:String = "activate";
    public static const ADDED:String    = "added";
    // Remaining constants omitted for brevity

    // read-only properties accessible through get functions
    function get bubbles():Boolean;
    function get currentTarget():EventDispatcher;
    function get eventPhase():uint;
    function get target():EventDispatcher;
    function get type():String;

    // class methods
    override function clone():Event;
    function isDefaultPrevented():Boolean;
    function preventDefault():Void;
    function stopImmediatePropagation():Void;
```

```
    function stopPropagation():Void;
    override function toString():String;
}

  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;
  }

}
```

The constructor is useful mainly for advanced ActionScript programmers who create and dispatch custom event objects. It is useful for our present discussion, however, to notice that the `type` parameter is the only required parameter. This means that every event object must have a type associated with it. The Flash Player API defines about four dozen event types, the details of which can be found in the API documentation for the Event class and its subclasses.

The EventPhase class defines three constant properties that represent the three phases of the event flow: `CAPTURING_PHASE`, `AT_TARGET`, and `BUBBLING_PHASE`. These properties store, respectively, the integer values `1`, `2` and `3`. These are the same integer values that represent the set of possible values for the `eventPhase` property. Because every event object has the `eventPhase` property, you can examine any event object to determine that event object's current phase.

Some two dozen event types are associated with the Event class itself. For the sake of brevity, most of these constants, which are constants in the Event class, are not listed above. However, the 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 testing to see whether an event object is of a certain type, use the following code:

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

rather than:

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

The Flash Player API defines several subclasses of the Event class. Each of the subclasses provides additional properties and event types that are unique to that category of events. For example, the previous example uses the MouseEvent class, which defines several event types related to that input device, including, among others, the `MouseEvent.CLICK`, `MouseEvent.DOUBLE_CLICK`, `MouseEvent.MOUSE_DOWN`, and `MouseEvent.MOUSE_UP` event types.

The Event class properties

The Event class defines a number of read-only properties that provide important information about an event object. The `Event.type` property indicates which of the four dozen event types to which this event belongs. The `Event.target` property indicates the target node to which the event was dispatched. The `Event.currentTarget` property indicates the node that is currently processing its event listeners for this event object. While it may seem odd not to know which node is currently processing the event object you are examining, keep in mind that you can register to listen for an event on any display object in that event object's event flow, and you can use any object or function to process that event object. As a project increases in size and complexity, the `Event.currentTarget` property will become more and more useful.

The remaining pair of read-only properties deserve special attention. The `Event.bubbles` property is an important property to take into account when establishing where to place your event listeners for a particular event. This property stores a Boolean value that determines whether the event object participates in the all three of the event flow phases or only the capture and target phases. If the value is `true`, then the event will participate in all three phases. If the value is `false`, then the event object will not participate in the bubbling phase.

The `Event.cancelable` property indicates whether the event type has an associated default behavior that can be prevented. The easiest way to describe a default behavior is through an example. The `textInput` event type is generated when a user enters a character into a TextField object in the Flash Player window. When Flash Player detects such an event, it creates an event of type `textInput` and dispatches that event into the display list hierarchy. If that event object passes through the display list without interruption—you will learn how to intercept an event object in the following section—then Flash Player will carry out the default behavior associated with that event type. In the case of the textInput event, the default behavior is that the character will be displayed in the TextField object.

Not all event types have associated default behaviors. 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 only associated with event objects dispatched by Flash Player, and do not exist for events dispatched programmatically through ActionScript. For example, you can use the methods of the EventDispatcher class to dispatch an event object of type `textInput`, but if you do there will not be 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.

The Event class methods

The `clone()` method allows you to create copies of an event object while the `toString()` method allows you to generate a string that includes all the properties of an event object along with their values. Both of these methods are meant to be overridden by subclasses to include any additional properties added by the subclasses.

The four other methods of the Event class fall into two categories. Two of the methods provide a means to stop an event object from continuing its journey through the display list. The two remaining methods deal with the cancellation of default behavior associated with an event.

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, on the other hand, 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.

> **Note:** A call to either `Event.stopPropogation()` or `Event.stopImmediatePropogation()` will not prevent default behavior from occurring.

To cancel the default behavior associated with an event, call the `Event.preventDefault()` method. This 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 `Event.cancelable` property of the event object. You can check whether an event's default behavior has already been cancelled by another event listener by calling the `Event.isDefaultPrevented()` method, which returns a Boolean value of `true` if the behavior has already been cancelled and `false` otherwise.

Event listeners

In previous versions of ActionScript, event listeners could only be registered directly with the object that generated the event. In the ActionScript 3.0 event model, you can register an event listener with any display list object that is part of an event object's event flow. This means that as long as the type of event you are listening for participates in the capture or bubbling phases, you can register an event listener with either the target node, or one of the target node's ancestor nodes.

Creating an event listener is a two step process. First, you create a function or class method that will execute in response to the event. This is sometimes called the listener function. Second, you register that function or method with a display list object that lies along the appropriate event flow.

Just as in ActionScript 2.0, the listener function that you create can be either a function defined outside of a class, or it can be a method of a class. What you name your function depends on how you choose to register your listener function using the `addEventListener()` method. One of that method's required parameters is called `listener`, and you use that parameter to register your listener function. You can name the function anything you like, as long as it does not conflict with an ActionScript reserved word.

Given that you can use a function or a class method, there are two ways you can pass in a listener function to `addEventListener()`:

- listener function defined outside of a custom class;
- listener function defined as part of a custom class (a class method);

These two techniques were also available to users of components in ActionScript 2.0 through the UIEventDispatcher class, with one important change to the meaning of the `this` keyword when passing in a class method. Examples of both options follow, along with details of what `this` means in each case.

Listener function defined outside of a class

The following code creates a listener function that listens for click events and registers the listener function with a display list object named `myDisplayObject`:

```
// a custom listener function
function myListenerFunction (evtObj:Event) {
  trace ("myListenerFunction() detected event: " + evtObj.type);
  trace ("this refers to: " + this);
}

// register your function with myDisplayObject
myDisplayObject.addEventListener("click", myListenerFunction);
```

When a user interacts with the resulting SWF file by clicking on the display list object named `myDisplayObject`, Flash Player will generate the following trace output:

```
myListenerFunction() detected event: click
this refers to: [object global]
```

Notice that the event object is passed as a parameter to myListenerFunction. This allows your listener function to examine the event object. In this case, the example uses 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 defines an event listener function as part of a custom class. A function defined as part of a custom class is usually called a method of that class.

This particular example focuses on a listener function with a custom name that is passed to the `addEventListener()` method.

This example requires the creation of an external ActionScript class file named CustomListener.as, as well as the creation of a display list object named `myDisplayObject`.

```
// Contents of the CustomListener.as file
class CustomListener extends DisplayObject{
  function myListenerMethod(evtObj:Event) {
    trace("myListenerMethod() detected event: " + evtObj.type);
    trace("this refers to: " + this);
  }
}

// ActionScript code
var myCustomListener:CustomListener = new CustomListener();
myDisplayObject.addEventListener(flash.events.MouseEventType.CLICK,
  myCustomListener.myListenerMethod);
```

If you call `addEventListener()` outside of your custom class, the second parameter to `addEventListener()` must contain the fully qualified method name. You must include the name of the instance (`myCustomListener`) as well as the name of the custom method (`myEventListener()`). This is not necessary if you call `addEventListener()` from the constructor or a method of your custom class.

When a user interacts with the resulting SWF file by clicking on the display list object named `myDisplayObject`, Flash Player will generate the following trace output:

```
myListenerMethod() detected event: click
this refers to: [object CustomListener]
```

Note that the `this` keyword refers to the CustomListener object. This is a change in behavior from ActionScript 2.0. User of components in ActionScript 2.0 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 listener object that contains the listener method. This was a significant issue for some programmers because it meant that they could not access other methods and properties of that listener object. 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, and as a result, the `this` keyword refers to the listener object `myCustomListener` and the programmer has access to the other methods and properties of the CustomListener 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 listener function, but it is not recommended. It is discussed here so that you know to avoid it. This technique is not recommended because the `this` keyword will refer to the global object instead of your listener object.

The following technique is not recommended:

```
var myListenerObject:Object = new Object();
myListenerObject.myListener = function (evtObj:Event) {
  trace("myListener has detected event: " + evtObj.type);
  trace(this);
}
myDisplayObject.addEventListener(flash.events.MouseEventType.CLICK,
  myListenerObject.myListener);
```

The results of the trace will look like this:

```
myListener has detected event: click
[object global]
```

You would expect that `this` would refer to `myListenerObject`, but instead it refers to the global object. When you pass in an arbitrary property name as a parameter 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 `myListenerObject` instance.

The EventDispatcher class

The EventDispatcher class is a base class that provides important event model functionality for every object on the display list. Recall that the DisplayObject class inherits from the EventDispatcher class. This means that any object on the display list has access to the methods of the EventDispatcher class.

Although the name EventDispatcher 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.

```
package flash.events
  {
    public class EventDispatcher implements IEventDispatcher
    {
       function EventDispatcher (target:IEventDispatcher=null);

       function addEventListener(eventName:String,
                                 listener:Object,
                                 useCapture:Boolean=false,
                                 priority:Integer=0):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 constructor is useful for advanced programmers who are creating custom events and dispatching them. You may notice that the EventDispatcher class implements the IEventDispatcher interface. This allows programmers who are creating custom classes that cannot inherit from EventDispatcher or one of its subclasses to instead implement the IEventDispatcher interface to gain access to its methods. If you do this, you should refer to the API documentation for the IEventDispatcher interface for implementation details.

The `addEventListener()` method is the workhorse of this class. You use it to register your event listeners. The two required parameters are `eventName` and `listener`. The `eventName` parameter allows you to specify the type of event and the `listener` parameter allows you to specify the listener function that will execute when the event occurs. The `listener` parameter can be either a reference to a function or class method.

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, then your listener will be active during the capture phase of the event flow. If `useCapture` is set to false, then 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 Events Model. It is provided in ActionScript 3.0 to provide programmers more flexibility in organizing their 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 lower 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.

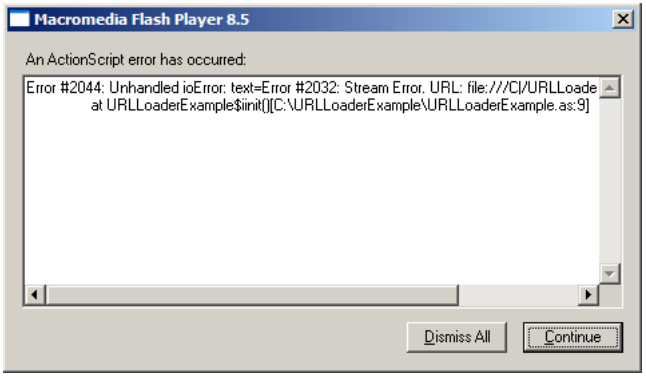
You can use the `removeEventListener()` method to remove an event listener that you no longer need. It is a good idea to remove any listeners that will no longer be used. Required parameters include the `eventName` and `listener` parameters, which are the same as the required parameters for the `addEventListener()` 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`, then again with it set to `false`.

The `dispatchEvent()` method can be used by advanced programmers to dispatch a custom event object into the event flow.

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

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 the stand-alone Flash Player.



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 property named "level" (`StatusEvent.level` and `NetStatusEvent.info.level`). When the value of the `level` property is `"error"`, then 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 standalone players, as a message in the output panel in the authoring player, and as an entry in the log file for Flex. It will not manifest at all in the release versions of Flash Player.

CHAPTER 20

Networking and Communication

20

The flash.net package contains classes to send and receive data across the Internet, such as to load content from remote URLs, and connect to remote Web Services. In previous versions of ActionScript, many of the classes within this package were top-level classes.

This chapter includes the following sections:

- Working with external data
- Connecting to other player instances
- Socket connections
- Working with files
- Application security

Working with external data

When you build large ActionScript projects, you often need to communicate with server side scripts, or load data from external XML or text files. This behavior 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, similar to how the LoadVars class worked in previous versions of ActionScript.

To load and parse a remote XML document, you use the URLLoader and URLRequest classes to load the remote XML document, then you can parse the XML document using the XML class's constructor, 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 URLVariables class

Flash Player 8.5 and ActionScript 3.0 replace the LoadVars class with URLLoader and URLVariables classes. The URLLoader class downloads data from a URL as ASCII or binary data. 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 ActionScript 3.0's advanced event handling model which allows you to listen for such events as: `complete`, `httpStatus`, `ioError`, `open`, `progress`, and `securityError`, which is a significant improvement over ActionScript 2.0's support for `LoadVars.onData`, `LoadVars.onHTTPStatus`, and `LoadVars.onLoad` event handlers.

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` event to be dispatched. The loaded data is decoded from UTF-8 or UTF-16 into a String.

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).

Loading data from external documents

When you build dynamic applications with ActionScript 3.0, you want 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 would retrieve a random tip from a database and save it to a text file once a day. Then your ActionScript application can load the contents of 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 seen in the following code:

```
var request:URLRequest = new URLRequest("params.txt");
request.method = URLRequestMethod.POST;
```

The external document which is loaded at runtime, params.txt, 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 gets parsed as strings, which can be split into an array using the `String.split()` method.

TIP	Avoid using reserved words as variable names in external data files as it can make reading and debugging your code a lot more difficult.

Once the data has loaded, the `Event.COMPLETE` event dispatches and the contents of the external document are available to use in the URLLoader's `data` property. This is seen in the following code:

```
private function onComplete(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:

```
private function onComplete(event:Event):void {
  var loader2:URLLoader = URLLoader(event.target);
  var variables:URLVariables = 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, such as:

```
var monthNameArray:Array = variables.dayNames.split(",");
```

TIP	If you are loading numeric data from external text files, you would need convert the values into numeric values using a top-level function such as `int()`, `uint()`, or `Number()`.

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 response. This is useful 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 whether the user supplied credentials are valid.

The following snippet creates a URLVariables object named `variables`, which creates a new variable called `name` in the URLVariables object. 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.

After the `URLLoader.load()` method completes and the `Event.COMPLETE` event dispatches, you can parse the server response by looking at the `URLLoader.data` property and creating a new URLVariables object:

```
var loader2:URLLoader = URLLoader(event.target);
```

The following code contains the contents of the 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 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 standalone player. This allows you to build very versatile applications which 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 classes that call out of the player environment such as the `fscommand()` method. You use this class when you want to pass data between SWF files running in the same domain. 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, without Flash Player.)

The simplest way to use a LocalConnection object is to allow communication only between LocalConnection objects located in the same domain because you won't have security issues. However, if you need to allow communication between domains, you have several ways to implement security measures.

TIP	It is possible to use LocalConnection objects to send and receive data within a single SWF file, but this is not recommended by Adobe. Instead, you should use SharedObjects.

There are three ways to add callbacks to your LocalConnection objects:

- Create a dynamic class that extends LocalConnection and dynamically attach methods.
- Subclass the LocalConnection class and add methods.
- Set the `LocalConnection.client` property to an object which implements the methods.

The first way to add callbacks, creating a dynamic class and dynamically attaching methods, is very similar to using the LocalConnection class in previous versions of ActionScript, as shown in the following code:

```
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 second way to add callbacks, subclassing the LocalConnection class, lets you define the methods within the custom class instead of dynamically adding them to the LocalConnection instance. This is demonstrated in the following code:

```
package {
```

```
  import flash.net.LocalConnection;
  import flash.util.trace;
  public class DynamicLocalConnection extends LocalConnection {
    public function DynamicLocalConnection(connectionName:String) {
      try {
        connect(connectionName);
      } catch(error:Error) {
        // 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:DynamicLocalConnection;
serverLC = new DynamicLocalConnection("serverName");
```

The third way to add callbacks, using the `LocalConnection.client` property, involves creating an Object and creating callbacks for the desired methods, as shown in the following code:

```
var callbackHandler:Object = new Object();
callbackHandler.onMethod = onMethod;
var serverLC:LocalConnection = new LocalConnection();
serverLC.client = callbackHandler;
try {
  serverLC.connect("serverName");
} catch (error:Error) {
  // ignore errors
}
...
private function onMethod():void {
  trace("doOnMethod");
}
```

The `LocalConnection.client` property indicates what object callback methods should be invoked. The default is `this`. You can use this 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.

Sending messages between two Flash Players

The local connection you create using the LocalConnection class lets you to communicate between different instances of Flash Player. For example, you could have multiple Flash Player insttances 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 class called DynamicLocalConnection that you can dynamically add methods and properties added to:

```
dynamic class DynamicLocalConnection extends flash.net.LocalConnection {}
```

To create a new instance of the custom DynamicLocalConnection class and add some callback functions, you can then use the following code:

```
var connection:DynamicLocalConnection = new DynamicLocalConnection();
connection.onMethod = this.onMethod;
connection.onQuit = this.onQuit;
connection.onFullScreen = this.onFullScreen;
```

Whenever a Flash Player instance connects to this SWF and tries to invoke either the `onMethod()` method, the `onQuit()` method, or `onFullScreen()` method, the requests will be sent to the specified methods within the class shown in the following code:

```
public function onMethod(params:String = null):void {
  trace("onMethod called from a client: " + params + "\n");
}
public function onQuit():void {
  trace("quitting in 5 seconds.");
  var quitTimer:Timer = new Timer(5000, 1);
  connection.close();
  quitTimer.addEventListener(TimerEvent.TIMER, onClosePlayer);
  quitTimer.start();
}
private function onClosePlayer(event:TimerEvent):void {
  fscommand("quit");
}
public function onFullScreen():void {
  fscommand("fullscreen", "true");
}
```

To create a LocalConnection sever, call the `LocalConnection.connect()` method and provide a unique connection name. If you already have a connection with the specified name, an ArgumentError is thrown saying that the connect failed because the object is already connected. The following snippet demonstrates how to create a new socket connection with the name `myLC`:

```
try {
  connection.connect("myLC");
} catch (e:Error) {
```

```
  trace("Error! Server already exists\n");
}
```

TIP	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 thrown if the name has already been used.

Connecting to the primary SWF from a secondary SWF requires that you create a new LocalConnection object in the sending LocalConnection object and call the `LocalConnection.send()` command 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("myLC", "onQuit");
```

This code connects to an existing LocalConnection with the connection name `myLC` and invokes the `onQuit()` 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 seen in the following snippet:

```
sendingConnection.send("myLC", "onMethod", (getTimer()/1000).toFixed(3) + "
  seconds");
```

Connecting to SWF documents in different domains

If you implement communication only between SWF files in the same domain, you can specify a `connectionName` 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` that begins with an underscore (_), which makes the SWF 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 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, the 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.

To only allow communications from certain domains, you call the `allowDomain()` or `allowInsecureDomain()` method of the LocalConnection class and pass a list of one or more domains which are allowed to access this LocalConnection object.

In earlier versions of ActionScript, the `LocalConnection.allowDomain` and `LocalConnection.allowInsecureDomain` were callback methods that had to be implemented by developers, and had to return a Boolean value. In ActionScript 3.0, the `LocalConnection.allowDomain()` and `LocalConnection.allowInsecureDomain()` methods 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 special domain, `*`, allows access from all domains, whereas 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 may not also have access to the local filesystem. 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` (flash.events.SecurityErrorEvent) dispatches. You can specify the caller's domain in the receiver's `LocalConnection.allowDomain()` method to work around this error.

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 to 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 the client and server don't need to exchange XML packets specifically. Instead, the connection can transfer data as binary information which allows you to connect to a wide range of services including mail servers (POP3, SMTP, and IMAP), and news (NNTP) servers.

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 events happens:

- 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).

Socket class

Introduced in ActionScript 3.0, the Socket class enables ActionScript to make socket connections, and read and write raw binary data. It is similar to the XMLSocket class, but does not dictate the format of the received/transmitted data, and is useful for interoperating with servers that use binary protocols. By using binary socket connections, you can write code that allows you to interact with several different Internet protocols (such as POP3, SMTP, IMAP, and NNTP), which enables Flash Player to connect to mail and news servers.

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 text:

```
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 machine or network, it doesn't need to be placed within the webroot 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 hostname and port number:

```
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 (`DataEventType.DATA`) is dispatched:

```
xmlsock.addEventListener(DataEventType.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:

```
smlsock.send(xmlFormattedData);
```

TIP The `XMLSocket.send()` method does not return a value that indicates whether the data was successfully transmitted.

NOTE Each message you send to the XML socket server must be terminated by a newline (\n).

Working with files

The FileReference class lets you add the ability to upload and download files between a client and server. Your users can upload or download files between their computer and a server. Users are prompted to select a file to upload or a location for download in 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. On the Macintosh, there is also a property for the file's creator type.

You can create an instance of the FileReference class in two ways. You can use the following `new` operator:

```
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 a file 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 the `FileReference.browse()` method or `FileReferenceList.browse()` method has been called and the user has selected a file from the file picker or until the `FileReference.download()` method has been used to select a file from the file picker.

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.

NOTE

The creator property is supported on the Macintosh only, all other platforms return `null`.

FileReference instances can be created in two ways:

- When you use the new operator with the FileReference constructor, such as:
  ```
  var myFileReference:FileReference = new FileReference();
  ```
- When you call `FileReferenceList.browse()`, which creates an array (named `fileList`) 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

Only one browse() or download() action can be performed at a time, because only one dialog box can be open at any point.

The server script that handles the file upload should expect a 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.

- The binary contents of the file.

Here is a sample HTTP POST request:

```
Content-Type: multipart/form-data; boundary=AaB03x
--AaB03x
Content-Disposition: form-data; name="Filedata"; filename="example.jpg"
Content-Type: application/octet-stream
... contents of example.jpg ...
--AaB03x--
```

Uploading files to a server

Using the `browse()` method of either the FileReference or FileReferenceList classes, users can select one or more files to upload to a server. Next, the `FileReference.upload()` method uploads a single file to the server. If multiple files were selected using the `FileReferenceList.browse()` method, an array of selected files would be created, `FileReferenceList.fileList`, and the `FileReference.upload()` method would be used to upload each file individually.

By default, the OS file picker dialog allows you to pick any file type from your local computer, although you can specify one or more custom file type filters 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);
```

Once the user has selected their files and clicked the Open button in the OS file picker, the `Event.SELECT` event is dispatched. If you browsed for a file using the `FileReference.browse()` method, you would upload the selected file using the following code:

```
var fileRef:FileReference = new FileReference();
fileRef.addEventListener(Event.SELECT, onSelect);
fileRef.browse();
private function onSelect(event:Event):void {
  var file:FileReference = FileReference(event.target);
  file.upload("http://localhost/fileUploadScript.cfm");
}
private function onComplete(event:Event):void {
  trace("uploaded");
}
```

If an input/output error occurs while the player is reading, writing, or transmitting the file, an `IOErrorEvent.IO_ERROR` event gets dispatched to the Flash Player. If `FileReference.upload()` attempts to upload a file to a server outside the caller's security sandbox, Flash Player will dispatch a `SecurityErrorEvent.SECURITY_ERROR` event. To prevent getting security sandbox errors, make sure you have a cross domain policy file on the remote server which you are connecting to.

TIP	Only Flash applications running in a browser using the browser plug-in or ActiveX control can provide a dialog to prompt the user to enter a username and password for authentication, and then only for downloads. For uploads using the plug-in or ActiveX control or upload/download using either the standalone or external players, 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 and overwrites any file with the same name. This sample code shows the bare minimum code necessary to accept a file upload, and shouldn't be used in a production environment. Ideally, you need to add data validation to ensure users upload accepted file types, such as an image instead of a potentially dangerous server-side script.

The following code demonstrates file uploads using PHP, and includes data validation. The script limits the number of uploaded files in the upload directory to ten, ensures that the file is less than 200 KB, and only permits 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;
  }
}
?>
```

Downloading files from a server

You can let users download files from a server using the `FileReference.download()` method, which takes two parameters: `url` and `defaultFileName`. The first parameter is the URL of the file to download, and the second parameter is optional and lets you specify a default file name that appears in the download file dialog box. If you omit the second parameter, `defaultFileName`, the file name from the specified URL is used.

The following code downloads a file named index.xml that is in the same directory as the SWF document:

```
var fileToDownload:FileReference = new FileReference();
fileToDownload.download("index.xml");
```

If you want to rename the file from index.xml to currentnews.xml, specify the `defaultFileName` parameter, as seen in the following snippet:

```
var fileToDownload:FileReference = new FileReference();
fileToDownload.download("index.xml", "currentnews.xml");
```

Renaming a file can be very useful if the server filename was unintuitive or server-generated. Another good reason to explicitly specify the `defaultFileName` parameter is 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 had 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. The code below demonstrates a ColdFusion script, download.cfm, that downloads one of two files on 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('../robots.txt')#"
  deletefile="No" />
  </cfcase>
  <cfdefaultcase>
    <cfcontent type="text/plain" file="#ExpandPath('../urlvars.txt')#"
  deletefile="No" />
  </cfdefaultcase>
</cfswitch>
```

The following ActionScript 3.0 snippet downloads a document based on which URL parameters are passed to a ColdFusion script:

```
var fileToDownload:FileReference = new FileReference();
fileToDownload.download("download.cfm?id=2", "somefile.txt");
```

If you only specify the `url` parameter in the previous snippet, Flash Player tries to save the document to the user's hard drive as download.cfm, which is probably not what you want. You specify the `defaultFileName` to repopulate the filename in the download file dialog box with the string `somefile.txt`.

FileReferenceList class

The FileReferenceList class lets 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 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();
fileRefList.browse(new Array(imageFilter, textFilter));
```

When browsing for one or more files using the FileReferenceList class, the process is a bit more difficult then browsing for a file using FileReference.browse(). Uploading multiple files means you have to loop over the array of selected files and upload each file individually using `FileReference.upload()`, as seen in the following code:

```
var fileRefList:FileReferenceList = new FileReferenceList();
fileRefList.addEventListener(Event.SELECT, onSelect);
fileRefList.browse();
private function onSelect(event:Event):void {
  var i:uint;
  var file:FileReference;
  var files:FileReferenceList = FileReferenceList(event.target);
  var selectedFileArray:Array = files.fileList;
  for (i=0; i<selectedFileArray.length; i++) {
    file = FileReference(selectedFileArray[i]);
    file.addEventListener(Event.COMPLETE, onComplete);
    file.upload("http://localhost/fileUploadScript.cfm");
  }
}
private function onComplete(event:Event):void {
  trace("uploaded");
}
```

Because the `Event.COMPLETE` event is added to each individual FileReference object in the array, Flash Player will call the `onComplete()` method when each individual file finishes uploading.

Application security

Flash Player 8 has made enhancements to the security model, in which Flash applications and SWF files on a local computer are not allowed to communicate with both the Internet and the local file system by default. When you develop a Flex or Flash application, you must indicate whether a SWF file is allowed to communicate with a network or with a local file system.

In Flash Player 7 and earlier, local SWF files could interact with other SWF files and load data from any remote or local computer without configuring security settings. In Flash Player 8 and later, a SWF file cannot make connections to the local file system and the network (such as the Internet) in the same application without making a security setting. This is for your safety, so a SWF file cannot read files on your hard disk and then send the contents of those files across the Internet.

This security restriction affects all locally deployed content, whether it is legacy content (a SWF file published for Flash Player 7 or earlier) or content created for Flash Player 8 or later. For example, if you use the Flash MX 2004 authoring tool, you could test a Flash application in a browser that runs locally and also accesses the Internet. In Flash Player 8.5, the same application asks the user for permission to communicate with the Internet.

When you test a file on your hard disk, there are several steps to determine whether the file is a local trusted (safe) document or a potentially untrusted (unsafe) document.

In Flash Player 8.5, local SWF files can have three levels of permission:

- Access the local file system only (the default level). The local SWF file can read from the local file system and universal naming convention (UNC) network paths and cannot communicate with the Internet.
- Access the network only. The local SWF file can access the network only (such as the Internet) and not the local file system where the SWF file is installed.
- Access to both the local file system and the network. The local SWF file can read from the local file system where the file is installed, read from and write to any server that grants it permission, and can cross-script other SWF files on either the network or the local file system that grant it permission.

CHAPTER 21

Client System Environment

21

flash.system package

The system package contains the ApplicationDomain, Capabilities, IME, Security, and System classes. These classes deal with client settings that might affect your application in Flash Player. The system package also includes the IMEConversionMode and SecurityPanel classes which contain static constants which can be used with the IME and Security classes respectively.

System class

Example: Resizing the stage to fit the screen

Capabilities class

ApplicationDomain class

The ApplicationDomain class is used as a container for discrete groups of class definitions. Application domains are used to partition classes that are in the same security domain. This allows for multiple definitions of the same class to exist and lets children reuse parent definitions.

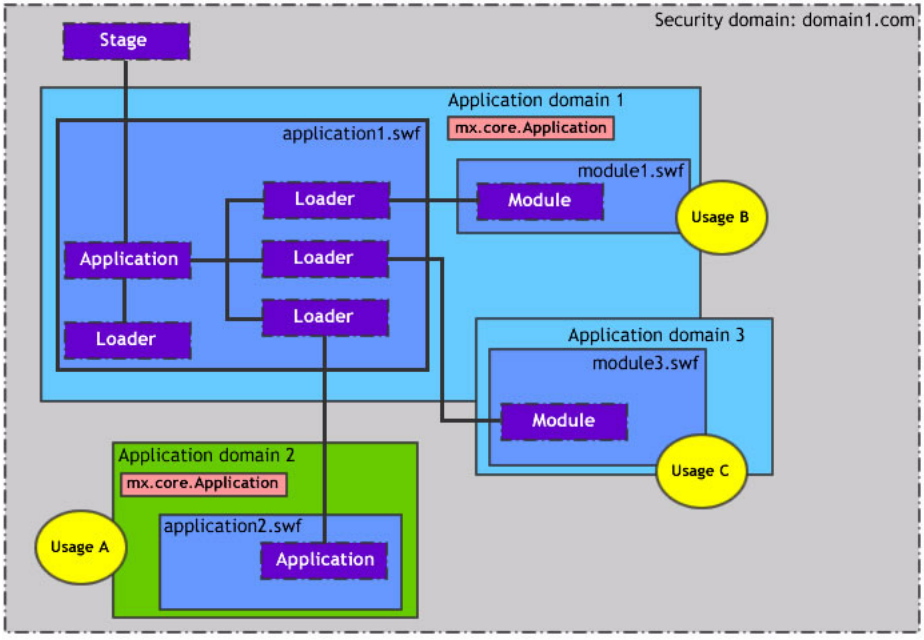
Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

Application domains are used when an external SWF is loaded through the Loader class API. All ActionScript 3.0 definitions contained in the loaded class are stored in the application domain, which is specified by the applicationDomain property of the URLRequest object (which you set) or LoaderInfo object (whose applicationDomain property is read-only).

All code in a SWF is defined to exist in an ApplicationDomain. 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 of 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 SWFs. In this scenario, the current domain is Application domain 1.

Usage A: Partition the child SWF 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 could be to have an old application dynamically loading a newer version of the same application, without conflict because the same class names are used but partitioned into different application domains.

The code to create an application domain that is a child of the system domain is as follows:

```
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 runtime shared library of the main application. The loaded SWF is treated as an RSL. Use this technique to load RSLs by a preloader before the application starts.

The code to set an application domain to the current domain is as follows:

```
request.url = "module1.swf";
request.digest = ".";
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 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 backwards compatible, and you ensure that the loading application is always newer than the things it loads, the children will use the parent versions. Having a new ApplicationDomain 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 code to create a new child domain of the current domain is as follows:

```
request.url = "module3.swf";
request.applicationDomain = new
  ApplicationDomain(ApplicationDomain.currentDomain);
```

Public Beta 1 Public Beta 1 Public Beta 1 Public Beta

IME class

An input field can be associated with a particular IME context. When you switch between input fields you can switch the IME as well, between hiragana, full-width numbers, half-width numbers, direct input, and so on.

Example: Switching the IME for data entry

CHAPTER 22

Using the External API

22

The ExternalInterface class is the External API, an application programming interface that enables straightforward communication between ActionScript and the Flash Player container; for example, an HTML page with JavaScript, or a desktop application with Flash Player embedded.

The use of ExternalInterface class is recommended for all communication between JavaScript and ActionScript, and serves as a replacement for the older `fscommand()` method. Using JavaScript on the HTML page, you can call an ActionScript function in Flash Player. The ActionScript function can return a value, and JavaScript receives it immediately as the return value of the call.

This chapter includes the following sections:

- The flash.external package
- Sample: Creating interaction between two applications
- Sample: Serializing data with ExternalInterface

The flash.external package

The flash.external package lets you communicate with the Flash Player container using ActionScript code. For example, if you embed an SWF file in an HTML page, that HTML page is the container. You would be able to communicate with the HTML page using the ExternalInterface class and JavaScript. Embedding a SWF file within an HTML page allows you to communicate with the Web browser which lets you integrate ActionScript 3.0 with technologies like AJAX.

About the External API

The ExternalInterface class (also called the External API) is a subsystem that lets you easily communicate from ActionScript and the Flash Player container to an HTML page with JavaScript or to any desktop application that embeds a Flash Player.

NOTE

This functionality replaces the older ActionScript `fscommand()` function (which can be found in the flash.system package) for interoperating with a HTML page or a container application. The External API offers more robust functionality than `fscommand()` in this situation.

The ExternalInterface class is available only under the following circumstances:

- 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 standalone 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 arguments.
- 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).

You can call an ActionScript function from JavaScript on an HTML page.

Using the External Interface class

The External Interface class includes one static property, and two static methods:

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 calling any of the methods in the ExternalInterface class, you should always check to make sure that the current container supports external interface communication.

NOTE

The `available` property reports if the type of container supports ExternalInterface connectivity, it will not tell you if JavaScript is enabled in the current browser.

The `ExternalInterface.addCallback()` method allows you to register a function which can be called from your container (HTML/JavaScript or desktop application).

The `ExternalInterface.call()` method executes code in a container surrounding the Flash Player. If the container is an HTML page, this method invokes a JavaScript function in a <script> element. If the container is some other ActiveX container, this method fires the `FlashCall` ActiveX event with the specified name, and the container processes the event. If the call failed, or the container method did not specify a return value, `null` is returned. The `call()` method will throw a `SecurityError` exception if the containing environment belongs to a security sandbox to which the calling code does not have access. This may be worked around by setting an appropriate value for `allowScriptAccess` in the containing environment (usually using the `allowScriptAccess` OBJECT/EMBED parameter in HTML).

Using the External API with an HTML container

One of the most common usages for the External API is to allow ActionScript applications to communicate with a Web browser. This allows ActionScript methods to call code written in JavaScript and vice versa.

Due to 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, your SWF document should always call the HTML page to say that the SWF document is ready to accept connections.

The following ActionScript code shows how to check if the External API is available, and if so, add a callback and notify the HTML container that it is ready to accept connections:

```
if (!ExternalInterface.available) {
  addText("ExternalInterface is not available in this container.");
} else {
```

```
    try {
      ExternalInterface.addCallback("addText", addText);
      addText("Callback added.");
      ExternalInterface.call("init");
    } catch (error:SecurityError) {
      addText("Security error: Unable to connect to external interface");
    }
}
```

The `addText()` function appends a specified string onto the end of a StringBuilder instance, `strBuilder`, then sets a text field's text property to the contents of the `strBuilder` object:

```
private function addText(str:String):void {
  strBuilder.append(str + "\n");
  tf.text = strBuilder.toString();
}
```

Now, once the SWF document has loaded it will call the JavaScript `init()` method to notify the HTML container that the SWF document is ready to be used.

The following code shows the code necessary to connect to the SWF document using JavaScript, and call the SWF document's `addText()` function:

```
<script language="JavaScript" type="text/javascript">
<!--
  function init() {
    var myMovie = thisMovie('ExternalInterfaceHTMLExample');
    myMovie.addText("message from JavaScript @ " + new Date());
  }
  function thisMovie(movieName) {
    if (navigator.appName.indexOf("Microsoft") != -1) {
      return window[movieName];
    } else {
      return document[movieName];
    }
  }
// -->
</script>
```

The JavaScript `init()` function connects to the SWF document with a name of "ExternalInterfaceHTMLExample", calls the `addText()` function in the SWF document and passes a simple string and date as a parameter.

WARNING

Due to differences in how browsers access content, you should always detect which browser the user is running and access the movie accordingly, using the `window` or `document` object.

Sample: Creating interaction between two applications

The following examples demonstrate how to embed a SWF document with a C# application using Visual Studio .NET and ActionScript 3.0.

To create the Windows application:

1. Open Visual Studio .NET.
2. Select File > New > Project from the main menu.
3. Select Visual C# Projects > Windows Application to create a standalone application.
4. Click OK to create the new project.
5. Make sure the Toolbox is open (select View > Toolbox).
6. Drag the Macromedia Flash Player ActiveX control from the Toolbox onto your project's form. If the Macromedia Flash Player ActiveX control isn't visible in your Toolbox follow the following steps:
   a. Open the Windows Forms section of the Toolbox.
   b. Right-click the Windows Form section and select Add/Remove Items from the pop-up menu. This opens the Customize Toolbox window.
   c. Select the COM Components tab, which lists all of the available COM components on your computer, including the Macromedia Flash Player ActiveX control.
   d. Scroll down to Shockwave Flash Object and select it. If this item is not listed, make sure that the Macromedia Flash Player ActiveX control is installed on your system.
   e. After you select the item, click OK to add the "Shockwave Flash Object" button to the Windows Forms section of the Toolbar.
7. Give the Shockwave Flash Object the name **flashControl** using the Properties window.
8. Add two text boxes to your form and give them the following names: **flashRequest_txt**, and **flashResponse_txt**.
9. Add a button to your form and give it the name **callFlash_btn**.
10. Add a label to your form and give it the name **label1**.
11. Modify the existing C# code to your application to match the following code (the changes to make appear in **boldface**):

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
```

```
using System.Data;
using System.IO;
namespace WindowsApplication1
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;
        private AxShockwaveFlashObjects.AxShockwaveFlash flashControl;
        private System.Windows.Forms.Button callFlash_btn;
        private System.Windows.Forms.TextBox flashRequest_txt;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.TextBox flashResponse_txt;
        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();
            String swfPath = Directory.GetCurrentDirectory() +
  Path.DirectorySeparatorChar + "extIntTest.swf";
            this.flashControl.LoadMovie(0, swfPath);
        }
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }
        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
```

```
            System.Resources.ResourceManager resources = new
System.Resources.ResourceManager(typeof(Form1));
            this.flashControl = new
AxShockwaveFlashObjects.AxShockwaveFlash();
            this.callFlash_btn = new System.Windows.Forms.Button();
            this.flashResponse_txt = new System.Windows.Forms.TextBox();
            this.flashRequest_txt = new System.Windows.Forms.TextBox();
            this.label1 = new System.Windows.Forms.Label();

((System.ComponentModel.ISupportInitialize)(this.flashControl)).Begin
Init();
            this.SuspendLayout();
            //
            // flashControl
            //
            this.flashControl.Enabled = true;
            this.flashControl.Location = new System.Drawing.Point(48,
224);
            this.flashControl.Name = "flashControl";
            this.flashControl.OcxState =
((System.Windows.Forms.AxHost.State)(resources.GetObject("flashContro
l.OcxState")));
            this.flashControl.Size = new System.Drawing.Size(200, 200);
            this.flashControl.TabIndex = 0;
            this.flashControl.FlashCall += new
AxShockwaveFlashObjects._IShockwaveFlashEvents_FlashCallEventHandler(
this.onFunctionCallFromFlash);
            //
            // callFlash_btn
            //
            this.callFlash_btn.Location = new System.Drawing.Point(408,
224);
            this.callFlash_btn.Name = "callFlash_btn";
            this.callFlash_btn.Size = new System.Drawing.Size(144, 23);
            this.callFlash_btn.TabIndex = 1;
            this.callFlash_btn.Text = "Call Flash";
            this.callFlash_btn.Click += new
System.EventHandler(this.onCallToFlashFunction);
            //
            // flashResponse_txt
            //
            this.flashResponse_txt.Location = new
System.Drawing.Point(384, 16);
            this.flashResponse_txt.Name = "flashResponse_txt";
            this.flashResponse_txt.Size = new System.Drawing.Size(200,
200);
            this.flashResponse_txt.TabIndex = 2;
            this.flashResponse_txt.Multiline = true;
            this.flashResponse_txt.Text = "";
            //
```

```
            // flashRequest_txt
            //
           this.flashRequest_txt.Location = new System.Drawing.Point(48,
16);
            this.flashRequest_txt.Size = new System.Drawing.Size(200,
200);
            this.flashRequest_txt.TabIndex = 3;
            this.flashRequest_txt.Name = "flashRequest_txt";
            this.flashRequest_txt.Multiline = true;
            this.flashRequest_txt.Text = "";
            //
            // label1
            //
            this.label1.Location = new System.Drawing.Point(392, 264);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(200, 64);
            this.label1.TabIndex = 4;
            this.label1.Text = "Call Flash External API";
            //
            // Form1
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(632, 453);
            this.Controls.Add(this.label1);
            this.Controls.Add(this.flashRequest_txt);
            this.Controls.Add(this.flashResponse_txt);
            this.Controls.Add(this.callFlash_btn);
            this.Controls.Add(this.flashControl);
            this.Name = "Form1";
            this.Text = "Form1";

((System.ComponentModel.ISupportInitialize)(this.flashControl)).EndIn
it();
            this.ResumeLayout(false);
        }
        #endregion
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
        private void onFunctionCallFromFlash(object sender,
AxShockwaveFlashObjects._IShockwaveFlashEvents_FlashCallEvent evt)
        {
            this.flashRequest_txt.Text = evt.request;
            this.flashControl.SetReturnValue("<string>String sent from
C#</string>");
```

```
        }
        private void onCallToFlashFunction(object sender,
    System.EventArgs evt)
        {
            String request = "<invoke
    name=\"testExternalAPI\"><arguments><string>foo</string></
    arguments><invoke>";
            try
            {
                this.flashResponse_txt.Text =
    this.flashControl.CallFunction(request);
            }
            catch(System.Exception e)
            {
                this.flashResponse_txt.Text = e.ToString();
            }
        }
    }
}
```

12. Save the C# project.
13. Build the solution (select Build > Build Solution from the main menu).

To create the SWF file:

1. Create a new ActionScript document and save it as **extIntTest.as** in the following C# application location: "..\WindowsApplication1\WindowsApplication1\bin\Debug\".
2. Add the following code to the ActionScript document:

```
package {
  import flash.display.Sprite;
  import flash.display.TextField;
  import flash.display.SimpleButton;
  import flash.external.ExternalInterface;
  import flash.events.*;

  public class extIntTest extends Sprite {
    public var callButton:SimpleButton;
    public var hostRequestParamTxt:TextField;
    public function extIntTest() {
      init();
    }
    private function init():void {
      hostRequestParamTxt = new TextField();
      hostRequestParamTxt.width = 200;
      hostRequestParamTxt.height = 160;
      hostRequestParamTxt.y = 40;
      hostRequestParamTxt.border = true;
      hostRequestParamTxt.background = true;
      hostRequestParamTxt.multiline = true;
```

```
            hostRequestParamTxt.wordWrap = true;
            addChild(hostRequestParamTxt);

            var downSprite:Sprite = new Sprite();
            downSprite.graphics.lineStyle(2, 0x202020);
            downSprite.graphics.beginFill(0x00FF00);
            downSprite.graphics.drawRect(10, 10, 80, 20);

            var upSprite:Sprite = new Sprite();
            upSprite.graphics.lineStyle(2, 0x202020);
            upSprite.graphics.beginFill(0xFFFF00);
            upSprite.graphics.drawRect(10, 10, 80, 20);

            callButton = new SimpleButton(upSprite, upSprite, downSprite,
        upSprite);
            callButton.useHandCursor = true;
            addChild(callButton);

            var success:Boolean =
        ExternalInterface.addCallback("testExternalAPI", testExternalAPI);
            hostRequestParamTxt.text = "added function:" + success;
            hostRequestParamTxt.text += "is available:" +
        ExternalInterface.available + "\n";
            callButton.addEventListener(MouseEvent.CLICK, onClick);
          }
          public function testExternalAPI(paramTxt:String):String {
            hostRequestParamTxt.text = paramTxt;
            return "Flash response to your request";
          }
          private function onClick(event:MouseEvent):void {
            hostRequestParamTxt.text =
        ExternalInterface.call("randomHostMethod", "Parameter from Flash");
          }
        }
    }
```

3. Save and compile the ActionScript document to generate a SWF file.

To run the application:

1. Select Debug > Start from the main menu in Visual Studio .NET, or run the EXE file that is in the following path: "..\WindowsApplication1\WindowsApplication1\bin\Debug".

   You can now see the bidirectional communication between the C# application and the SWF document. When you click the "Call Flash" button in the C# application, the application calls and receives a response from the embedded SWF file. Also, when you click the Button component in the SWF document the SWF file calls and receives a response from the C# application.

   Close collapsed procedure

Sample: Serializing data with ExternalInterface

The following examples demonstrate how to create and implement a serializer class in C#, which can be used to serialize data and pass it between C# and a SWF file. The serializer class has four public methods:

EncodeInvoke Encodes a function name and a C# ArrayList of arguments into XML that IShockwaveFlash.CallFunction expects.

EncodeResult Encodes a result value into XML that IShockwaveFlash.SetReturnValue expects.

DecodeInvoke Called when you receive a FlashCall. Requests are passed to the DecodeInvoke method, and it will decode the function name called and produce an ArrayList with the arguments in it.

DecideResult Called when IShockwaveFlash.CallFunction returns a result. This method decodes the received XML into a C# data type.

To create the Windows application:

1. Create a new C# class and save it as **ExternalInterfaceSerializer.cs**. This class can encode and decode the XML needed for ExternalInterface.
2. Add the following code to the C# class file:

```
using System;
using System.Collections;
using System.Xml;
using System.IO;
using System.Text;

namespace WindowsApplication_0927
{
  public class ExternalInterfaceSerializer
  {
    public string functionName;
    public ArrayList arguments;

    private void WriteArray(XmlTextWriter writer, ArrayList array)
    {
      writer.WriteStartElement("array");

      int index = 0;

      foreach (Object value in array)
      {
        writer.WriteStartElement("property");
```

```
        writer.WriteAttributeString("id", index.ToString());
        WriteElement(writer, value);
        writer.WriteEndElement();
        index++;
      }

      writer.WriteEndElement();
    }

    private void WriteObject(XmlTextWriter writer, Hashtable table)
    {
      writer.WriteStartElement("object");

      foreach (DictionaryEntry entry in table)
      {
        writer.WriteStartElement("property");
        writer.WriteAttributeString("id", entry.Key.ToString());
        WriteElement(writer, entry.Value);
        writer.WriteEndElement();
      }

      writer.WriteEndElement();
    }

    private void WriteElement(XmlTextWriter writer, Object value)
    {
      if (value == null)
      {
        writer.WriteStartElement("null");
        writer.WriteEndElement();
      }
      else if (value is string)
      {
        writer.WriteStartElement("string");
        writer.WriteString(value.ToString());
        writer.WriteEndElement();
      }
      else if (value is bool)
      {
        writer.WriteStartElement((bool)value ? "true" : "false");
        writer.WriteEndElement();
      }
      else if (value is Single || value is Double || value is int ||
value is uint)
      {
        writer.WriteStartElement("number");
        writer.WriteString(value.ToString());
        writer.WriteEndElement();
      }
      else if (value is ArrayList)
```

```
    {
      WriteArray(writer, (ArrayList)value);
    }
    else if (value is Hashtable)
    {
      WriteObject(writer, (Hashtable)value);
    }
    else
    {
      throw new Exception("Could not encode type of " + value);
    }
  }

  public string EncodeInvoke(string functionName, ArrayList arguments)
  {
    StringBuilder sb = new StringBuilder();

    XmlTextWriter writer = new XmlTextWriter(new StringWriter(sb));

    writer.WriteStartElement("invoke");
    writer.WriteAttributeString("name", functionName);
    writer.WriteAttributeString("returntype", "xml");

    writer.WriteStartElement("arguments");

    foreach (Object value in arguments)
    {
      WriteElement(writer, value);
    }

    writer.WriteEndElement();
    writer.WriteEndElement();

    writer.Flush();
    writer.Close();

    return sb.ToString();
  }

  public string EncodeResult(Object value)
  {
    StringBuilder sb = new StringBuilder();

    XmlTextWriter writer = new XmlTextWriter(new StringWriter(sb));

    WriteElement(writer, value);

    writer.Flush();
    writer.Close();
```

```
    return sb.ToString();
  }

  public void DecodeInvoke(string xml)
  {
    XmlTextReader reader = new XmlTextReader(xml,
XmlNodeType.Document, null);

    reader.Read();

    functionName = reader.GetAttribute("name");
    arguments = new ArrayList();

    reader.ReadStartElement("invoke");
    reader.ReadStartElement("arguments");

    while (!(reader.NodeType == XmlNodeType.EndElement && reader.Name
== "arguments"))
    {
      arguments.Add(ReadElement(reader));
    }

    reader.ReadEndElement();
    reader.ReadEndElement();
  }

  public Object DecodeResult(string xml)
  {
    XmlTextReader reader = new XmlTextReader(xml,
XmlNodeType.Document, null);
    reader.Read();
    return ReadElement(reader);
  }

  private ArrayList ReadArray(XmlTextReader reader)
  {
    ArrayList array = new ArrayList();

    while (!(reader.NodeType == XmlNodeType.EndElement && reader.Name
== "array"))
    {
      int id = int.Parse(reader.GetAttribute("id"));
      reader.ReadStartElement("property");
      array.Add(ReadElement(reader));
      reader.ReadEndElement();
    }

    return array;
  }
```

```
    private Hashtable ReadObject(XmlTextReader reader)
    {
      Hashtable table = new Hashtable();

      while (!(reader.NodeType == XmlNodeType.EndElement && reader.Name
== "object"))
      {
        string id = reader.GetAttribute("id");
        reader.ReadStartElement("property");
        table.Add(id, ReadElement(reader));
        reader.ReadEndElement();
      }

      return table;
    }

    private Object ReadElement(XmlTextReader reader)
    {
      if (reader.NodeType != XmlNodeType.Element)
      {
        throw new XmlException();
      }

      if (reader.Name == "true")
      {
        reader.Read();
        return true;
      }

      if (reader.Name == "false")
      {
        reader.Read();
        return false;
      }

      if (reader.Name == "null" || reader.Name == "undefined")
      {
        reader.Read();
        return null;
      }

      if (reader.IsStartElement("number"))
      {
        reader.ReadStartElement("number");
        double value = Double.Parse(reader.Value);
        reader.Read();
        reader.ReadEndElement();
        return value;
      }
```

```
        if (reader.IsStartElement("string"))
        {
          reader.ReadStartElement("string");
          string value = reader.Value;
          reader.Read();
          reader.ReadEndElement();
          return value;
        }

        if (reader.IsStartElement("array"))
        {
          reader.ReadStartElement("array");
          ArrayList value = ReadArray(reader);
          reader.ReadEndElement();
          return value;
        }

        if (reader.IsStartElement("object"))
        {
          reader.ReadStartElement("object");
          Hashtable value = ReadObject(reader);
          reader.ReadEndElement();
          return value;
        }

        throw new XmlException();
      }

    }
}
```

3. Save the changes to the ExternalInterfaceSerializer class.
4. Create a new Windows application in Visual Studio .NET by following steps 1 through 10 of the previous example called "Creating the Windows Application".
5. Add the ExternalInterfaceSerializer.cs class to the current project.
6. Modify the existing C# code to your application to match the following code (the changes to make appear in **boldface**):

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
namespace WindowsApplication1
{
  /// <summary>
  /// Summary description for Form1.
```

```
/// </summary>
public class Form1 : System.Windows.Forms.Form
{
  /// <summary>
  /// Required designer variable.
  /// </summary>
  private ExternalInterfaceSerializer serializer = new
ExternalInterfaceSerializer();
  private System.ComponentModel.Container components = null;
  private AxShockwaveFlashObjects.AxShockwaveFlash flashControl;
  private System.Windows.Forms.Button callFlash_btn;
  private System.Windows.Forms.TextBox flashRequest_txt;
  private System.Windows.Forms.Label label1;
  private System.Windows.Forms.TextBox flashResponse_txt;
  public Form1()
  {
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
    String swfPath = Directory.GetCurrentDirectory() +
Path.DirectorySeparatorChar + "test.swf";
    this.flashControl.LoadMovie(0, swfPath);
  }
  /// <summary>
  /// Clean up any resources being used.
  /// </summary>
  protected override void Dispose( bool disposing )
  {
    if( disposing )
    {
      if (components != null)
      {
        components.Dispose();
      }
    }
    base.Dispose( disposing );
  }
  #region Windows Form Designer generated code
  /// <summary>
  /// Required method for Designer support - do not modify
  /// the contents of this method with the code editor.
  /// </summary>
  private void InitializeComponent()
  {
    System.Resources.ResourceManager resources = new
System.Resources.ResourceManager(typeof(Form1));
    this.flashControl = new
AxShockwaveFlashObjects.AxShockwaveFlash();
    this.callFlash_btn = new System.Windows.Forms.Button();
```

```
        this.flashResponse_txt = new System.Windows.Forms.TextBox();
        this.flashRequest_txt = new System.Windows.Forms.TextBox();
        this.label1 = new System.Windows.Forms.Label();

    ((System.ComponentModel.ISupportInitialize)(this.flashControl)).Begin
    Init();
        this.SuspendLayout();
        //
        // flashControl
        //
        this.flashControl.Enabled = true;
        this.flashControl.Location = new System.Drawing.Point(48, 224);
        this.flashControl.Name = "flashControl";
        this.flashControl.OcxState =
    ((System.Windows.Forms.AxHost.State)(resources.GetObject("flashContro
    l.OcxState")));
        this.flashControl.Size = new System.Drawing.Size(200, 200);
        this.flashControl.TabIndex = 0;
        this.flashControl.FlashCall += new
    AxShockwaveFlashObjects._IShockwaveFlashEvents_FlashCallEventHandler(
    this.onFunctionCallFromFlash);
        //
        // callFlash_btn
        //
        this.callFlash_btn.Location = new System.Drawing.Point(408, 224);
        this.callFlash_btn.Name = "callFlash_btn";
        this.callFlash_btn.Size = new System.Drawing.Size(144, 23);
        this.callFlash_btn.TabIndex = 1;
        this.callFlash_btn.Text = "Call Flash";
        this.callFlash_btn.Click += new
    System.EventHandler(this.onCallToFlashFunction);
        //
        // flashResponse_txt
        //
        this.flashResponse_txt.Location = new System.Drawing.Point(384,
    16);
        this.flashResponse_txt.Name = "flashResponse_txt";
        this.flashResponse_txt.Size = new System.Drawing.Size(200, 200);
        this.flashResponse_txt.TabIndex = 2;
        this.flashResponse_txt.Multiline = true;
        this.flashResponse_txt.Text = "";
        //
        // flashRequest_txt
        //
        this.flashRequest_txt.Location = new System.Drawing.Point(48,
    16);
        this.flashRequest_txt.Size = new System.Drawing.Size(200, 200);
        this.flashRequest_txt.TabIndex = 3;
        this.flashRequest_txt.Name = "flashRequest_txt";
        this.flashRequest_txt.Multiline = true;
```

```
        this.flashRequest_txt.Text = "";
        //
        // label1
        //
        this.label1.Location = new System.Drawing.Point(392, 264);
        this.label1.Name = "label1";
        this.label1.Size = new System.Drawing.Size(200, 64);
        this.label1.TabIndex = 4;
        this.label1.Text = "Call Flash External API";
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(632, 453);
        this.Controls.Add(this.label1);
        this.Controls.Add(this.flashRequest_txt);
        this.Controls.Add(this.flashResponse_txt);
        this.Controls.Add(this.callFlash_btn);
        this.Controls.Add(this.flashControl);
        this.Name = "Form1";
        this.Text = "Form1";

    ((System.ComponentModel.ISupportInitialize)(this.flashControl)).EndIn
    it();
        this.ResumeLayout(false);
      }
      #endregion
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
        Application.Run(new Form1());
      }

      private void onFunctionCallFromFlash(object sender,
    AxShockwaveFlashObjects._IShockwaveFlashEvents_FlashCallEvent evt)
      {
        // will produce the string: <string>String sent from C#</string>
        String returnValueString = serializer.EncodeResult("String sent
    from C#");

        String text = "Raw XML received:\r\n" + evt.request + "\r\n\r\n";

        serializer.DecodeInvoke(evt.request);

        text += "Function called:" + serializer.functionName + "\r\n";

        int index = 1;
```

```
        foreach (Object value in serializer.arguments)
        {
          text += "Argument #" + index + ": " + value + "\r\n";
          index++;
        }

        this.flashRequest_txt.Text = text;
        this.flashControl.SetReturnValue(returnValueString);
      }

      private void onCallToFlashFunction(object sender, System.EventArgs
    evt)
      {
        ArrayList arguments = new ArrayList();
        arguments.Add("foo");

        // will produce the string: <invoke
    name="testExternalAPI"><arguments><string>foo</string></arguments></
    invoke>
        String request = serializer.EncodeInvoke("testExternalAPI",
    arguments);
        try
        {
          String resultXML = this.flashControl.CallFunction(request);

          Object result = serializer.DecodeResult(resultXML);

          this.flashResponse_txt.Text = result.ToString();
        }
        catch(System.Exception e)
        {
          this.flashResponse_txt.Text = e.ToString();
        }
      }
    }
}
```

7. Save the C# project.
8. Build the solution (select Build > Build Solution from the main menu).

To create the SWF file:

1. Create a new ActionScript document and save it as **extIntTest.as** in the following C# application location: "..\WindowsApplication1\WindowsApplication1\bin\Debug\".
2. Add the following code to the ActionScript document:

```
package {
  import flash.display.Sprite;
  import flash.display.TextField;
  import flash.display.SimpleButton;
```

```
import flash.external.ExternalInterface;
import flash.events.*;

public class extIntTest extends Sprite {
  public var callButton:SimpleButton;
  public var hostRequestParamTxt:TextField;
  public function extIntTest() {
    init();
  }
  private function init():void {
    hostRequestParamTxt = new TextField();
    hostRequestParamTxt.width = 200;
    hostRequestParamTxt.height = 160;
    hostRequestParamTxt.y = 40;
    hostRequestParamTxt.border = true;
    hostRequestParamTxt.background = true;
    hostRequestParamTxt.multiline = true;
    hostRequestParamTxt.wordWrap = true;
    addChild(hostRequestParamTxt);

    var downSprite:Sprite = new Sprite();
    downSprite.graphics.lineStyle(2, 0x202020);
    downSprite.graphics.beginFill(0x00FF00);
    downSprite.graphics.drawRect(10, 10, 80, 20);

    var upSprite:Sprite = new Sprite();
    upSprite.graphics.lineStyle(2, 0x202020);
    upSprite.graphics.beginFill(0xFFFF00);
    upSprite.graphics.drawRect(10, 10, 80, 20);

    callButton = new SimpleButton(upSprite, upSprite, downSprite,
upSprite);
    callButton.useHandCursor = true;
    addChild(callButton);

    var success:Boolean =
ExternalInterface.addCallback("testExternalAPI", testExternalAPI);
    hostRequestParamTxt.text = "added function:" + success;
    hostRequestParamTxt.text += "is available:" +
ExternalInterface.available + "\n";
    callButton.addEventListener(MouseEvent.CLICK, onClick);
  }
  public function testExternalAPI(paramTxt:String):String {
    hostRequestParamTxt.text = paramTxt;
    return "Flash response to your request";
  }
  private function onClick(event:MouseEvent):void {
    hostRequestParamTxt.text =
ExternalInterface.call("randomHostMethod", "Parameter from Flash");
  }
```

```
        }
    }
```

3. Save and compile the ActionScript document to generate a SWF file.

CHAPTER 23

Printing

23

Note: In this draft of *Programming ActionScript 3.0,* this chapter includes only heading titles, with no further content. The following chapters include more complete information:

Flash Player tasks and system printing

Creating a PrintJob instance

Setting size, scale, and orientation

Example: Multiple page printing

Example: Scaled and cropped printing

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

X