MX

macromedia®
FLASH™MX 2004

Using ActionScript in Flash

Acknowledgments

Director: Erick Vera

Project Management: Julee Burdekin, Erick Vera

Writing: Jay Armstrong, Jody Bleyle, Mary Burger, Francis Cheng, Jen deHaan, Stephanie Gowin, Phillip Heinz, Shimul Rahim, Samuel R. Neff

Managing Editor: Rosana Francescato

Editing: Linda Adler, Mary Ferguson, Mary Kraemer, Noreen Maher, Antonio Padial, Lisa Stanziano, Anne Szabla

Production Management: Patrice O'Neill

Media Design and Production: Adam Barnett, Christopher Basmajian, Aaron Begley, John Francis

Second Edition: June 2004

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

INTRODUCTION
Getting Started with ActionScript

Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 are the professional standard authoring tools for producing high-impact web experiences. ActionScript is the language you use to add interactivity to a Flash applications, whether your applications are simple animated movies or more complex rich Internet applications. You don't have to use ActionScript to use Flash, but if you want to provide basic or complex user interactivity, work with objects other than those built into Flash (such as buttons and movie clips), or otherwise turn a SWF file into a more robust user experience, you'll probably want to use ActionScript.

Intended audience

This manual assumes that you have already installed Flash MX 2004 or Flash MX Professional 2004 and know how to use it. You should know how to place objects on the Stage and manipulate them in the Flash authoring environment. If you have written programs before, ActionScript will seem familiar. But if you're new to programming, ActionScript isn't hard to learn. It's easy to start with simple commands and build more complexity as you progress.

System requirements

ActionScript does not have any system requirements in addition to Flash MX 2004 or Flash MX Professional 2004. To use ActionScript 2.0, you must use Flash MX 2004.

The documentation assumes that you are using the default publishing settings for your Flash files: Flash Player 7 and ActionScript 2.0. If you change either of these settings, explanations and code samples shown in the documentation may not work correctly. If you develop applications for earlier versions of Flash Player, see "Porting existing scripts to Flash Player 7" on page 13 or Appendix D, "Writing Scripts for Earlier Versions of Flash Player," on page 319.

Using the documentation

This manual provides an overview of ActionScript syntax, information on how to use ActionScript when working with different types of objects, and details on the syntax and usage of every language element. The following list summarizes the contents of this manual.

- Chapter 1, “What’s New in Flash MX 2004 ActionScript,” on page 11 describes features that are new in ActionScript, changes to the compiler and debugger, and the new programming model for the ActionScript 2.0 language.
- Chapter 2, “ActionScript Basics,” on page 23 describes the terminology and basic concepts used in the rest of the manual.
- Chapter 3, “Using Best Practices,” on page 65 helps explain the best practices for using Flash and writing ActionScript.
- Chapter 4, “Writing and Debugging Scripts,” on page 139 describes the ActionScript editor and debugger within Flash that makes it easier to write code.
- Chapter 5, “Handling Events,” on page 167 discusses a few different ways to handle events: event handler methods, event listeners, and button and movie clip event handlers.
- Chapter 6, “Creating Interaction with ActionScript,” on page 179 describes some simple ways in which you can create more interactive applications, including controlling when SWF files play, creating custom pointers, and creating sound controls.
- Chapter 7, “Using the Built-In Classes,” on page 197 lists the built-in classes in ActionScript and provides a brief overview of how you use them to access powerful features in ActionScript.
- Chapter 8, “Working with Movie Clips,” on page 205 describes movie clips and the ActionScript you can use to control them.
- Chapter 9, “Working with Text,” on page 221 describes the different ways you can control text in Flash and include information on text formatting.
- Chapter 10, “Creating Custom Classes with ActionScript 2.0,” on page 247 describes how to create custom classes and objects for manipulating data in your applications.
- Chapter 11, “Working with External Data,” on page 275 describes how to process data from external sources using server- or client-side scripts in your applications.
- Chapter 12, “Working with External Media,” on page 295 describes how to import external media files such as JPEG, MP3, and other SWF files in your Flash applications.
- Appendix A, “Error Messages,” on page 305 contains a list of error messages that the Flash compiler can generate.
- Appendix B, “Deprecated Flash 4 operators,” on page 311 lists all the ActionScript operators and their associativity.
- Appendix C, “Keyboard Keys and Key Code Values,” on page 313 lists all the keys on a standard keyboard and the corresponding ASCII key code values that are used to identify the keys in ActionScript.
- Appendix D, “Writing Scripts for Earlier Versions of Flash Player,” on page 319 provides guidelines to help you write scripts that are syntactically correct for the player version you are targeting.
- Appendix E, “Object-Oriented Programming with ActionScript 1” provides information on using the ActionScript 1 object model to write scripts.

This manual explains how to use the ActionScript language. For information on the language elements themselves, see ActionScript Language Reference Help.

Before writing your own scripts, you should complete the lessons "Write Scripts with ActionScript" and "Create a Form with Conditional Logic and Send Data," which provide a hands-on introduction to working with ActionScript. To find these lessons, select Help > How Do I > Quick Tasks.

When you find information about a certain command you want to use, you can look up its entry in ActionScript Language Reference Help.

When you find information about a certain command you want to use, you can look up its entry in *Flash ActionScript Language Reference.*

Typographical conventions

The following typographical conventions are used in this manual:

- `Code font` indicates ActionScript code.
- *`Code font italic`* indicates an element, such as an ActionScript parameter or object name, that you replace with your own text when writing a script.

Terms used in this document

The following terms are used in this manual:

- *You* refers to the developer who is writing a script or application.
- *The user* refers to the person who is running your scripts and applications.
- *Compile time* is the time at which you publish, export, test, or debug your document.
- *Runtime* is the time at which your script is running in Flash Player.

ActionScript terms such as *method* and *object* are defined in Chapter 2, "ActionScript Basics," on page 23.

Additional resources

Specific documentation about Flash and related products is available separately.

- For information about the elements that comprise the ActionScript language, see ActionScript Language Reference Help.
- For information about working in the Flash authoring environment, see Using Flash Help.
- For information about working with components, see Using Components Help.
- For information about creating communication applications with Flash Communication Server, see *Developing Communications Applications* and *Managing Flash Communication Server.*
- For information about accessing web services with Flash applications, see *Using Flash Remoting*.

You can find application FLA files that use common Flash functionality installed with Flash. These applications were designed to introduce new Flash developers to the capabilities of Flash applications as well as show advanced developers how Flash features work in context. On Windows, you can find the applications in \Program Files\Macromedia\Flash MX 2004\ Examples. On the Macintosh, you can find the applications in HD/Applications/Macromedia Flash MX 2004/Examples.

The Macromedia DevNet website (www.macromedia.com/devnet) is updated regularly with the latest information on Flash, plus advice from expert users, advanced topics, examples, tips, and other updates. Check the website often for the latest news on Flash and how to get the most out of the program.

The Macromedia Flash Support Center (www.macromedia.com/support/flash) provides TechNotes, documentation updates, and links to additional resources in the Flash community.

CHAPTER 1
What's New in Flash MX 2004 ActionScript

Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 provide several enhancements that make it easy for you to write robust scripts using the ActionScript language. The new features, which are discussed in this chapter, include new language elements, improved editing and debugging tools (see "ActionScript editor changes" on page 20 and "Debugging changes" on page 21), and the introduction of a more object-oriented programming model (see "New object-oriented programming model" on page 21).

This chapter also contains an extensive section that you should read carefully if you plan to publish any of your existing Flash MX or earlier files to Flash Player 7 (see "Porting existing scripts to Flash Player 7" on page 13).

Updating Flash XML files

It is important that you always have the latest Flash XML files installed. Macromedia sometimes introduces features in dot releases (minor releases) of Flash Player. When such a release is available, you should update your version of Flash to get the latest XML files. Otherwise, the Flash MX 2004 compiler might generate errors if you use new properties or methods that were unavailable in the version of Flash Player that came with your Flash installation.

For example, Flash Player 7 (7.0.19.0) contained a new method for the System object, `System.security.loadPolicyFile`. To access this method, you must use the Player Updater installer to update all the Flash Players that are installed with Flash. Otherwise, the Flash MX 2004 compiler displays errors.

Remember that you can install a Player Updater that is one or more major versions ahead of your version of Flash. By doing this, you will get the XML files that you need but shouldn't have any compiler errors when you publish to older versions of Flash Player. Sometimes new methods or properties are available to older versions, and having the latest XML files minimizes the compiler errors you get when you try to access older methods or properties.

New and changed language elements

This section describes the ActionScript language elements that are new or changed in Flash MX 2004. To use any of these elements in your scripts, you must target Flash Player 7 (the default) when you publish your documents.

- The `Array.sort()` and `Array.sortOn()` methods let you add parameters to specify additional sorting options, such as ascending and descending sorting, whether to consider case-sensitivity when sorting, and so on.
- The `Button.menu`, `MovieClip.menu`, and `TextField.menu` properties work with the new ContextMenu and ContextMenuItem classes to let you associate context menu items with Button, MovieClip, or TextField objects.
- The "ContextMenu class" and "ContextMenuItem class" let you customize the context menu that appears when a user right-clicks (Windows) or Control-clicks (Macintosh) in Flash Player.
- The "Error class" and the `throw` and `try..catch..finally` commands let you implement more robust exception handling.
- The `LoadVars.addRequestHeader()` and `XML.addRequestHeader()` methods add or change HTTP request headers (such as `Content-Type` or `SOAPAction`) sent with `POST` actions.
- The `MMExecute()` function lets you issue Flash JavaScript API commands from ActionScript.
- The `Mouse.onMouseWheel` event listener is generated when the user scrolls using the mouse wheel (Windows only).
- The `MovieClip.getNextHighestDepth()` method lets you create MovieClip instances at runtime and be guaranteed that their objects render in front of the other objects in a parent movie clip's *z*-order space.
- The `MovieClip.getInstanceAtDepth()` method lets you access dynamically created MovieClip instances using the depth as a search index.
- The `MovieClip.getSWFVersion()` method lets you determine which version of Flash Player is supported by a loaded SWF file.
- The `MovieClip.getTextSnapshot()` method and the "TextSnapshot object" let you work with text that is in static text fields in a movie clip.
- The `MovieClip._lockroot` property lets you specify that a movie clip will act as `_root` for any movie clips loaded into it or that the meaning of `_root` in a movie clip won't change if that movie clip is loaded into another movie clip.
- The "MovieClipLoader class" lets you monitor the progress of files as they are loaded into movie clips.
- The "NetConnection class" and "NetStream class" let you stream local Flash video (FLV) files.
- The "PrintJob class" gives you (and the user) more control over printing from Flash Player.
- The `Sound.onID3` event handler provides access to ID3 data associated with a Sound object that contains an MP3 file.
- The `Sound.id3` property provides access to the metadata that is part of an MP3 file.
- The "System class" has new objects and methods, and the "System.capabilities object" has several new properties.

- The `TextField.condenseWhite` property lets you remove extra white space from HTML text fields that are rendered in a browser.
- The `TextField.mouseWheelEnabled` property lets you specify whether a text field's contents should scroll when the mouse pointer is positioned over a text field and the user rolls the mouse wheel.
- The "TextField.StyleSheet class" lets you create a style sheet object that contains text formatting rules such as font size, color, and other formatting styles.
- The `TextField.styleSheet` property lets you attach a style sheet object to a text field.
- The `TextFormat.getTextExtent()` method accepts a new parameter, and the object it returns contains a new member.
- The `XML.addRequestHeader()` method lets you add or change HTTP request headers (such as `Content-Type` or `SOAPAction`) sent with `POST` actions.

New security model and legacy SWF files

Rules for how Flash Player determines whether two domains are the same have changed in Flash Player 7. In addition, rules that determine whether and how a SWF file served from an HTTP domain can access a SWF file or load data from an HTTPS domain have changed. In most cases, these changes won't affect you unless you are porting your existing SWF files to Flash Player 7.

However, if you have SWF files published for Flash Player 6 or earlier that load data from a file stored on a server, and the calling SWF file is playing in Flash Player 7, the user might see a dialog box that didn't appear before, asking whether to allow access. You can prevent this dialog box from appearing by implementing a *policy file* on the site where the data is stored. For more information on this dialog box, see "About compatibility with previous Flash Player security models" on page 294.

You might also need to implement a policy file if you are using runtime shared libraries. If either the loading or loaded SWF file is published for Flash Player 7 and the loading and loaded files aren't served from the exact same domain, use a policy file to permit access. For more information on policy files, see "About allowing cross-domain data loading" on page 290.

Porting existing scripts to Flash Player 7

As with any new release, Flash Player 7 supports more ActionScript commands than previous versions of the player; you can use these commands to implement more robust scripts. (See "New and changed language elements" on page 12.)However, if you used any of these commands in your existing scripts, the script might not work correctly if you publish it for Flash Player 7.

For example, if you have a script with a function named `Error`, the script might appear to compile correctly but might not run as expected in Flash Player 7, because `Error` is now a built-in class (making it a reserved word) in ActionScript. You can fix your script by renaming the Error function to something else, such as `ErrorCondition`. For a complete list of reserved words, see "Keywords and reserved words" on page 32.

Also, Flash Player 7 implements several changes that affect how one SWF file can access another SWF file, how external data can be loaded, and how local settings and data (such as privacy settings and locally persistent shared objects) can be accessed. Finally, the behavior of some existing features has changed.

If you have existing scripts written for Flash Player 6 or earlier that you want to publish for Flash Player 7, you might need to modify the scripts so they conform with the implementation of Flash Player 7 and work as designed. These modifications are discussed in this section under the following headings:

ECMA-262 compliance

Several changes were implemented in Flash Player 7 to conform more closely to the ECMA-262 Edition 3 standard (see www.ecma-international.org/publications/standards/Ecma-262.htm). In addition to the class-based programming techniques available in ActionScript 2.0 (see "New object-oriented programming model" on page 21), other features have been added and certain behaviors have changed. Also, when publishing for Flash Player 7 and using ActionScript 2.0, you can cast one object type to another. For more information, see "Casting objects" on page 42. These capabilities don't require you to update existing scripts; however, you might want to use them if you publish your scripts to Flash Player 7 and then continue to revise and enhance them.

Unlike the changes mentioned above, the changes listed in the following table (some of which also improve ECMA compliance) can cause existing scripts to work differently than they did before. If you used these features in existing scripts that you want to publish to Flash Player 7, review the changes to make sure your code still works as intended or to determine whether you need to rewrite your code. In particular, because `undefined` is evaluated differently in certain cases, you should initialize all variables in scripts that you port to Flash Player 7.

SWF file published for Flash Player 7	SWF file published for earlier versions of Flash Player
Case-sensitivity is enforced (variable names that differ only in capitalization are interpreted as being different variables). This change also affects files loaded with `#include` and external variables loaded with `LoadVars.load()`. For more information, see "Case sensitivity" on page 28.	Case-sensitivity is not supported (variable names that differ only in capitalization are interpreted as being the same variable).

Evaluating `undefined` in a numeric context returns `NaN`.
`myCount +=1;`
`trace(myCount); // NaN`
	Evaluating `undefined` in a numeric context returns 0.
	`myCount +=1;`
	`trace(myCount); // 1`

SWF file published for Flash Player 7	SWF file published for earlier versions of Flash Player

When `undefined` is converted to a string, the result is `undefined`.
`firstname = "Joan ";`
`lastname = "Flender";`
`trace(firstname + middlename + lastname);`
`// Joan undefinedFlender`
	When `undefined` is converted to a string, the result is an empty string ("").
	`firstname = "Joan ";`
	`lastname = "Flender";`
	`trace(firstname + middlename + lastname);`
	`// Joan Flender`

When you convert a string to a Boolean value, the result is `true` if the string has a length greater than zero; the result is `false` for an empty string.	When you convert a string to a Boolean value, the string is first converted to a number; the result is `true` if the number is nonzero, `false` otherwise.

When setting the length of an array, only a valid number string sets the length. For example, "6" works but " 6" or "6xyz" does not.
`my_array=new Array();`
`my_array[" 6"] ="x";`
`trace(my_array.length); // 0`
`my_array["6xyz"] ="x";`
`trace(my_array.length); // 0`
`my_array["6"] ="x";`
`trace(my_array.length); // 7`
	When setting the length of an array, even a malformed number string sets the length.
	`my_array=new Array();`
	`my_array[" 6"] ="x";`
	`trace(my_array.length); // 7`
	`my_array["6xyz"] ="x";`
	`trace(my_array.length); // 7`
	`my_array["6"] ="x";`
	`trace(my_array.length); // 7`

Domain-name rules for settings and local data

In Flash Player 6, superdomain matching rules are used by default when accessing local settings (such as camera or microphone access permissions) or locally persistent data (shared objects). That is, the settings and data for SWF files hosted at here.xyz.com, there.xyz.com, and xyz.com are shared and are all stored at xyz.com.

In Flash Player 7, exact-domain matching rules are used by default. That is, the settings and data for a file hosted at here.xyz.com are stored at here.xyz.com, the settings and data for a file hosted at there.xyz.com are stored at there.xyz.com, and so on.

A new property, `System.exactSettings`, lets you specify which rules to use. This property is supported for files published for Flash Player 6 or later. For files published for Flash Player 6, the default value is `false`, which means superdomain matching rules are used. For files published for Flash Player 7, the default value is `true`, which means exact-domain matching rules are used.

If you use settings or persistent local data and want to publish a Flash Player 6 SWF file for Flash Player 7, you might need to set this value to `false` in the ported file.

For more information, see `System.exactSettings` in Flash ActionScript Language Reference Help.

Cross-domain and subdomain access between SWF files

When you develop a series of SWF files that communicate with each other—for example, when using `loadMovie()`, `MovieClip.loadMovie()`, `MovieClipLoader.LoadClip()`, or Local Connection objects—you might host the SWF files in different domains or in different subdomains of a single superdomain.

In files published for Flash Player 5 or earlier, there were no restrictions on cross-domain or subdomain access.

In files published for Flash Player 6, you could use the `LocalConnection.allowDomain` handler or `System.security.allowDomain()` method to specify permitted cross-domain access (for example, to let a file at someSite.com be accessed by a file at someOtherSite.com), and no command was needed to permit subdomain access (for example, a file at www.someSite.com could be accessed by a file at store.someSite.com).

Files published for Flash Player 7 implement access between SWF files differently from earlier versions in two ways. First, Flash Player 7 implements exact-domain matching rules instead of superdomain matching rules. Therefore, the file being accessed (even if it is published for a Player version earlier than Flash Player 7) must explicitly permit cross-domain or subdomain access; this topic is discussed in this section. Second, a file hosted at a site using a secure protocol (HTTPS) must explicitly permit access from a file hosted at a site using an insecure protocol (HTTP or FTP); this topic is discussed in the next section (see "HTTP to HTTPS protocol access between SWF files" on page 18).

The following table summarizes domain-matching rules in different versions of Flash Player:

Files published for	Cross-domain access between SWF files	Subdomain access between SWF files
Flash Player 5 or earlier	No restrictions	No restrictions
Flash Player 6	Superdomain matching	No restrictions
Flash Player 7	Exact domain matching Explicit permission for HTTPS-hosted files to access HTTP- or FTP-hosted files	Exact domain matching Explicit permission for HTTPS-hosted files to access HTTP- or FTP-hosted files

Because Flash Player 7 implements exact-domain matching rules instead of superdomain matching rules, you might have to modify existing scripts if you want to access them from files that are published for Flash Player 7. (You can still publish the modified files for Flash Player 6.) If you used any `LocalConnection.allowDomain()` or `System.security.allowDomain()` statements in your files and specified superdomain sites to permit, you must change your parameters to specify exact domains instead. The following code shows an example of changes you might have to make:

```
// Flash Player 6 commands in a SWF file at www.anyOldSite.com
// to allow access by SWF files that are hosted at www.someSite.com
// or at store.someSite.com
System.security.allowDomain("someSite.com");
my_lc.allowDomain = function(sendingDomain) {
  return(sendingDomain=="someSite.com");
}
// Corresponding commands to allow access by SWF files
// that are published for Flash Player 7
System.security.allowDomain("www.someSite.com", "store.someSite.com");
my_lc.allowDomain = function(sendingDomain) {
  return(sendingDomain=="www.someSite.com" ||
    sendingDomain=="store.someSite.com");
}
```

You might also have to add statements such as these to your files if you aren't currently using them. For example, if your SWF file is hosted at www.someSite.com and you want to allow access by a SWF file published for Flash Player 7 at store.someSite.com, you must add statements such as the following example to the file at www.someSite.com (you can still publish the file at www.someSite.com for Flash Player 6):

```
System.security.allowDomain("store.someSite.com");
my_lc.allowDomain = function(sendingDomain) {
  return(sendingDomain=="store.someSite.com");
}
```

In addition, consider that if a Flash Player 6 application running within Flash Player 7 tries to access data outside its exact domain, Flash Player 7 domain-matching rules are enforced and the user is prompted to allow or deny access.

To summarize, you might have to modify your files to add or change `allowDomain` statements if you publish files for Flash Player 7 that meet the following conditions:

- You implemented cross-SWF scripting (using `loadMovie()`, `MovieClip.loadMovie()`, `MovieClipLoader.LoadClip()`, or Local Connection objects).
- The called SWF file (of any version) is not hosted at a site using a secure protocol (HTTPS), or the calling and called SWF files are both hosted at HTTPS sites. (If only the called SWF file is HTTPS, see "HTTP to HTTPS protocol access between SWF files" on page 18.)
- The SWF files are not in same domain (for example, one file is at www.domain.com and one is at store.domain.com).

You must make the following changes:

- If the called SWF file is published for Flash Player 7, include `System.security.allowDomain` or `LocalConnection.allowDomain` in the called SWF file, using exact domain-name matching.
- If the called SWF file is published for Flash Player 6, modify the called file to add or change a `System.security.allowDomain` or `LocalConnection.allowDomain` statement, using exact domain-name matching, as shown in the code examples earlier in this section. You can publish the modified file for either Flash Player 6 or 7.
- If the called SWF file is published for Flash Player 5 or earlier, port the called file to Flash Player 6 or 7 and add a `System.security.allowDomain` statement, using exact domain-name matching, as shown in the code examples earlier in this section. (LocalConnection objects aren't supported in Flash Player 5 or earlier.)

For more information, see the tech note at www.macromedia.com/support/flash/ts/documents/security_sandbox.htm.

HTTP to HTTPS protocol access between SWF files

As discussed in the previous section, rules for cross-domain and subdomain access have changed in Flash Player 7. In addition to the exact-domain matching rules now being implemented, you must explicitly permit files hosted at sites using a secure protocol (HTTPS) to be accessed by files hosted at sites using an insecure protocol. Depending on whether the called file is published for Flash Player 7 or 6, you must implement either one of the `allowDomain` statements (see "Cross-domain and subdomain access between SWF files" on page 15), or use the new `LocalConnection.allowInsecure Domain` or `System.security.allowInsecureDomain()` statements.

Warning: Implementing an `allowInsecureDomain()` statement compromises the security offered by the HTTPS protocol. You should make these changes only if you can't reorganize your site so that all SWF files are served from the HTTPS protocol.

The following code shows an example of the changes you might have to make:

```
// Commands in a Flash Player 6 SWF file at https://www.someSite.com
// to allow access by Flash Player 7 SWF files that are hosted
// at http://www.someSite.com or at http://www.someOtherSite.com
System.security.allowDomain("someOtherSite.com");
my_lc.allowDomain = function(sendingDomain) {
  return(sendingDomain=="someOtherSite.com");
}
// Corresponding commands in a Flash Player 7 SWF file
// to allow access by Flash Player 7 SWF files that are hosted
// at http://www.someSite.com or at http://www.someOtherSite.com
System.security.allowInsecureDomain("www.someSite.com",
  "www.someOtherSite.com");
my_lc.allowInsecureDomain = function(sendingDomain) {
  return(sendingDomain=="www.someSite.com" ||
    sendingDomain=="www.someOtherSite.com");
}
```

You might also have to add statements such as these to your files if you aren't currently using them. A modification might be necessary even if both files are in same domain (for example, a file in http://www.domain.com is calling a file in https://www.domain.com).

To summarize, you might have to modify your files to add or change statements if you publish files for Flash Player 7 that meet the following conditions:

- You implemented cross-SWF scripting (using `loadMovie()`, `MovieClip.loadMovie()`, `MovieClipLoader.LoadClip()`, or Local Connection objects).
- The calling file is not hosted using an HTTPS protocol, and the called file is HTTPS.

You must make the following changes:

- If the called file is published for Flash Player 7, include `System.security.allowInsecureDomain` or `LocalConnection.allowInsecureDomain` in the called file, using exact domain-name matching, as shown in the code examples earlier in this section. This statement is required even if the calling and called SWF files are in same domain.

- If the called file is published for Flash Player 6 or earlier, and both the calling and called files are in same domain (for example, a file in http://www.domain.com is calling a file in https://www.domain.com), no modification is needed.
- If the called file is published for Flash Player 6, the files are not in same domain, and you don't want to port the called file to Flash Player 7, modify the called file to add or change a `System.security.allowDomain` or `LocalConnection.allowDomain` statement, using exact domain-name matching, as shown in the code examples earlier in this section.
- If the called file is published for Flash Player 6 and you want to port the called file to Flash Player 7, include `System.security.allowInsecureDomain` or `LocalConnection.allowInsecureDomain` in the called file, using exact domain-name matching, as shown in the code examples earlier in this section. This statement is required even if both files are in same domain.
- If the called file is published for Flash Player 5 or earlier, and both files are not in the same domain, you can do one of two things. You can either port the called file to Flash Player 6 and add or change a `System.security.allowDomain` statement, using exact domain-name matching, as shown in the code examples earlier in this section, or you can port the called file to Flash Player 7, and include a `System.security.allowInsecureDomain` statement in the called file, using exact domain-name matching, as shown in the code examples earlier in this section.

Server-side policy files for permitting access to data

A Flash document can load data from an external source by using one of the following data loading calls: `XML.load()`, `XML.sendAndLoad()`, `LoadVars.load()`, `LoadVars.sendAndLoad()`, `loadVariables()`, `loadVariablesNum()`, `MovieClip.loadVariables()`, `XMLSocket.connect()`, and Macromedia Flash Remoting (`NetServices.createGatewayConnection`). Also, a SWF file can import runtime shared libraries (RSLs), or assets defined in another SWF file, at runtime. By default, the data or RSL must reside in the same domain as the SWF file that is loading that external data or media.

To make data and assets in runtime shared libraries available to SWF files in different domains, you should use a *cross-domain policy file*. A cross-domain policy file is an XML file that provides a way for the server to indicate that its data and documents are available to SWF files served from certain domains, or from all domains. Any SWF file that is served from a domain specified by the server's policy file is permitted to access data or RSLs from that server.

If you are loading external data, you should create policy files even if you don't plan to port any files to Flash Player 7. If you are using RSLs, you should create policy files if either the calling or called file is published for Flash Player 7.

For more information, see "About allowing cross-domain data loading" on page 290.

ActionScript editor changes

The ActionScript editor has been updated in several ways to make it more robust and easier to use. These changes are summarized in this section.

Word wrapping You can now use the Options pop-up menu in the Script pane, Debugger panel, and Output panel to enable or disable word wrapping. You can also toggle word wrapping using the pop-up menu on the Actions panel. The keyboard shortcut is Control+Shift+W (Windows) or Command+Shift+W (Macintosh).

Viewing context-sensitive help When your pointer is positioned over an ActionScript language element in the Actions toolbox or in the Script pane, you can use the View Help item in the context menu to display a help page about that element.

Importing scripts When you select Import Script from the pop-up menu in the Actions panel, the imported script is copied into the script at the insertion point in your code file. In previous versions of Flash, importing a script overwrote the contents of the existing script.

Single-click breakpoints To add a debugging breakpoint before a line of code in the Debugger panel or the Script pane on the Actions panel, you can click in the left margin. In previous versions of Flash, clicking in the left margin selected a line of code. The new way to select a line of code is to Control-click (Windows) or Command-click (Macintosh).

Normal and expert modes no longer in Actions panel In previous versions of Flash, you could work in the Actions panel either in normal mode, in which you filled in options and parameters to create code, or in expert mode, in which you added commands directly into the Script pane. In Flash MX 2004 and Flash MX Professional 2004, you can work in the Actions panel only by adding commands directly to the Script pane. You can still drag commands from the Actions toolbox into the Script pane or use the Add (+) button above the Script pane to add commands to a script.

Pinning multiple scripts You can pin multiple scripts within a FLA file along the bottom of the Script pane in the Actions panel. In previous versions of Flash, you could pin only one script at a time.

Script navigator The left side of the Actions panel now contains two panes: the Actions toolbox and a new Script navigator. The Script navigator is a visual representation of the structure of your FLA file; you can navigate through your FLA file here to locate ActionScript code.

Integrated Script window for editing external files (Flash Professional only) You can use the ActionScript editor in a Script window (separate from the Actions panel) to write and edit external script files. Syntax coloring, code hinting, and other preferences are supported in the Script window, and the Actions toolbox is also available. To display the Script window, use File > New, and select the type of external file you want to edit. You can have multiple external files open at the same time; filenames appear on tabs across the top of the Script window. (The tabs appear only in Windows.)

Debugging changes

This section describes changes that improve your ability to debug your scripts.

Output window changed to Output panel You can now move and dock the Output panel in the same way as any other panel in Flash.

Improved error reporting at compile time In addition to providing more robust exception handling, ActionScript 2.0 provides several new compile-time errors. For more information, see Appendix A, "Error Messages," on page 305.

Improved exception handling The Error class and the `throw` and `try..catch..finally` commands let you implement more robust exception handling.

New object-oriented programming model

The ActionScript language has grown and developed since its introduction several years ago. With each new release of Flash, additional keywords, objects, methods, and other language elements have been added to the language. However, unlike earlier releases of Flash, Flash MX 2004 and Flash MX Professional 2004 introduce several new language elements that implement object-oriented programming in a more standard way than before. Because these language elements represent a significant enhancement to the core ActionScript language, they represent a new version of ActionScript: ActionScript 2.0.

ActionScript 2.0 is not a new language. Rather, it comprises a core set of language elements that make it easier to develop object-oriented programs. With the introduction of keywords such as `class`, `interface`, `extends`, and `implements`, ActionScript syntax is now easier to learn for programmers who are familiar with other languages. New programmers can learn more standard terminology that they can apply to other object-oriented languages they might study in the future.

The object-oriented programming (OOP) features in ActionScript 2.0 are based on the ECMAScript 4 Draft Proposal currently in development by ECMA TC39-TG1 (see www.mozilla.org/js/language/es4/index.html). Because the ECMA-4 proposal is not yet a standard, and because it is still changing, ActionScript 2.0 does not conform exactly to this specification.

ActionScript 2.0 supports all the standard elements of the ActionScript language; it lets you write scripts that more closely adhere to standards used in other object-oriented languages, such as Java. ActionScript 2.0 should be of interest primarily to intermediate or advanced Flash developers who are building applications that require the implementation of classes and subclasses. ActionScript 2.0 also lets you declare the object type of a variable when you create it (see "Strict data typing" on page 41) and provides significantly improved compiler errors (see Appendix A, "Error Messages," on page 305).

The following list shows the language elements that are new in ActionScript 2.0:

- `class`
- `extends`
- `implements`

- `interface`
- `dynamic`
- `static`
- `public`
- `private`
- `get`
- `set`
- `import`

Key facts about ActionScript 2.0 include the following points:

- Scripts that use ActionScript 2.0 to define classes or interfaces must be stored as external script files, with a single class defined in each script; that is, classes and interfaces cannot be defined in the Actions panel.
- You can import individual class files implicitly (by storing them in a location specified by global or document-specific search paths and then using them in a script) or explicitly (by using the `import` command); you can import packages (collections of class files in a directory) by using wildcards.
- Applications developed with ActionScript 2.0 are supported by Flash Player 6 and later.

Caution: The default publish setting for new files created in Flash MX 2004 is ActionScript 2.0. If you plan to modify an existing FLA file to use ActionScript 2.0 syntax, ensure that the FLA file specifies ActionScript 2.0 in its publish settings. If it does not, your file will compile incorrectly, although Flash will not generate compiler errors.

For more information on using ActionScript 2.0 to write object-oriented programs in Flash, see Chapter 10, "Creating Custom Classes with ActionScript 2.0," on page 247.

Although using ActionScript 2.0 is recommended, you can continue to use ActionScript 1 syntax, especially if you are doing more traditional Flash work such as simple animation that doesn't require user interaction.

CHAPTER 2
ActionScript Basics

ActionScript has rules of grammar and punctuation that determine which characters and words are used to create meaning and in which order they can be written. For example, in English, a period ends a sentence; in ActionScript, a semicolon ends a statement.

The general rules described in this section apply to all ActionScript. Most ActionScript terms also have individual requirements; for the rules for a specific term, see its entry in *Flash ActionScript Language Reference*. Applying the basics of ActionScript in a way that creates elegant programs can be a challenge for users who are new to ActionScript. For more information on how to apply the rules described in this section, see "Using Best Practices" on page 65.

This section contains the following topics:

Differences between ActionScript and JavaScript

ActionScript is similar to the core JavaScript programming language. You don't need to know JavaScript to use and learn ActionScript; however, if you know JavaScript, ActionScript will seem familiar.

This manual does not attempt to teach general programming. There are many resources that provide more information about general programming concepts and the JavaScript language.

- The European Computers Manufacturers Association (ECMA) document ECMA-262 is derived from JavaScript and serves as the international standard for the JavaScript language. ActionScript is based on the ECMA-262 specification. For more information, see www.ecma-international.org/publications/standards/Ecma-262.htm).
- Netscape DevEdge Online has a JavaScript Developer Central site (http://developer.netscape.com/tech/javascript/index.html) that contains documentation and articles useful for understanding ActionScript. The most valuable resource is the *Core JavaScript Guide.*
- The Java Technology site has tutorials on object-oriented programming (http://java.sun.com/docs/books/tutorial/java/index.html) that are targeted for the Java language but are useful for understanding concepts that you can apply to ActionScript.

Some of the differences between ActionScript and JavaScript are described in the following list:

- ActionScript does not support browser-specific objects such as Document, Window, and Anchor.
- ActionScript does not completely support all the JavaScript built-in objects.
- ActionScript does not support some JavaScript syntax constructs, such as statement labels.
- In ActionScript, the `eval()` function can perform only variable references.
- ActionScript 2.0, the latest version of the ActionScript language, supports several features that are not in the ECMA-262 specification, such as classes and strong typing. Many of these features are modeled after the ECMAScript 4 Draft Proposal currently in development by ECMA TC39-TG1 (see www.mozilla.org/js/language/es4/index.html), the standards committee in charge of the evolution of the ECMA-262 specification.
- ActionScript does not support regular expressions using the RegExp object. However, Macromedia Central does support the RegExp object.

Terminology

As with all scripting languages, ActionScript uses its own terminology. The following list provides an introduction to important ActionScript terms:

Boolean is a `true` or `false` value.

Classes are data types that you can create to define a new type of object. To define a class, you use the `class` keyword in an external script file (not in a script you are writing in the Actions panel).

Constants are elements that don't change. For example, the constant `Key.TAB` always has the same meaning: it indicates the Tab key on a keyboard. Constants are useful for comparing values.

Constructors are functions that you use to define (initialize) the properties and methods of a class. By definition, constructors are functions within a class definition that have the same name as the class. For example, the following code defines a Circle class and implements a constructor function:

```
// file Circle.as
class Circle {
  private var circumference:Number;
// constructor
  function Circle(radius:Number){
    this.circumference = 2 * Math.PI * radius;
  }
}
```

The term *constructor* is also used when you create (instantiate) an object based on a particular class. The following statements are calls to the constructor functions for the built-in Array class and the custom Circle class:

```
var my_array:Array = new Array();
var my_circle:Circle = new Circle(9);
```

Data types describe the kind of information a variable or ActionScript element can contain. The built-in ActionScript data types are String, Number, Boolean, Object, MovieClip, Function, null, and undefined. For more information, see "About data types" on page 34.

Events are actions that occur while a SWF file is playing. For example, different events are generated when a movie clip loads, the playhead enters a frame, the user clicks a button or movie clip, or the user types on the keyboard.

Event handlers are special actions that manage events such as `mouseDown` or `load`. There are two kinds of ActionScript event handlers: event handler methods and event listeners. (There are also two event handlers, `on()` and `onClipEvent()`, that you can assign directly to buttons and movie clips.) In the Actions toolbox, each ActionScript object that has event handler methods or event listeners has a subcategory called Events or Listeners. Some commands can be used both as event handlers and as event listeners and are included in both subcategories. For more information on event management, see "Handling Events" on page 167.

Expressions are any legal combination of ActionScript symbols that represent a value. An expression consists of operators and operands. For example, in the expression `x + 2`, `x` and `2` are operands and + is an operator.

Functions are blocks of reusable code that can be passed parameters and can return a value. For more information, see "Creating functions" on page 61.

Identifiers are names used to indicate a variable, property, object, function, or method. The first character must be a letter, underscore (_), or dollar sign ($). Each subsequent character must be a letter, number, underscore, or dollar sign. For example, `firstName` is the name of a variable.

Instances are objects that contain all the properties and methods of a particular class. For example, all arrays are instances of the Array class, so you can use any of the methods or properties of the Array class with any array instance.

Instance names are unique names that let you target instances you create, or movie clip and button instances on the Stage. For example, in the following code, "names" and "studentName" are instance names for two objects, an array and a string:

```
var names:Array = new Array();
var studentName:String = new String();
```

You use the Property inspector to assign instance names to instances on the Stage. For example, a master symbol in the library could be called `counter` and the two instances of that symbol in the SWF file could have the instance names `scorePlayer1_mc` and `scorePlayer2_mc`. The following code sets a variable called `score` inside each movie clip instance by using instance names:

```
this.scorePlayer1_mc.score = 0;
this.scorePlayer2_mc.score = 0;
```

You can use strict data typing when creating instances so that code hints (see "Using code hints" on page 147) appear as you type your code. For more information, see "Strictly typing objects to trigger code hints" on page 145.

Keywords are reserved words that have special meaning. For example, `var` is a keyword used to declare local variables. You cannot use a keyword as an identifier. For example, `var` is not a legal variable name. For a list of keywords, see "Keywords and reserved words" on page 32.

Methods are functions associated with a class. For example, `sortOn()` is a built-in method associated with the Array class. You can also create functions that act as methods, either for objects based on built-in classes or for objects based on classes that you create. For example, in the following code, `clear()` becomes a method of a `controller` object that you have previously defined:

```
function reset(){
  this.x_pos = 0;
  this.y_pos = 0;
}
controller.clear = reset;
controller.clear();
```

The following examples show how you create methods of a class:

```
//ActionScript 1 example
A = new Object();
A.prototype.myMethod = function() {
  trace("myMethod");
}

//ActionScript 2 example
class B {
  function myMethod() {
    trace("myMethod");
  }
}
```

Objects are collections of properties and methods; each object has its own name and is an instance of a particular class. Built-in objects are predefined in the ActionScript language. For example, the built-in Date class provides information from the system clock.

Operators are terms that calculate a new value from one or more values. For example, the addition (+) operator adds two or more values together to produce a new value. The values that operators manipulate are called *operands*.

Parameters (also called *arguments*) are placeholders that let you pass values to functions. For example, the following `welcome()` function uses two values it receives in the parameters `firstName` and `hobby`:

```
function welcome(firstName:String, hobby:String):String {
  var welcomeText:String = "Hello, " + firstName + ". I see you enjoy " + hobby
    +".";
  return welcomeText;
}
```

Packages are directories that contain one or more class files and reside in a designated classpath directory (see "Understanding the classpath" on page 268).

Properties are attributes that define an object. For example, `length` is a property of all arrays that specifies the number of elements in the array.

Statements are language elements that perform or specify an action. For example, the `return` statement returns a result as a value of the function in which it executes. The `if` statement evaluates a condition to determine the next action that should be taken. The `switch` statement creates a branching structure for ActionScript statements.

Target paths are hierarchical addresses of movie clip instance names, variables, and objects in a SWF file. You name a movie clip instance in the movie clip Property inspector. (The main Timeline always has the name `_root`.) You can use a target path to direct an action at a movie clip or to get or set the value of a variable or property. For example, the following statement is the target path to the `volume` property of the object named `stereoControl`:

```
stereoControl.volume
```

For more information on target paths, see "Using absolute and relative target paths" in *Using Flash*.

Variables are identifiers that hold values of any data type. Variables can be created, changed, and updated. The values they store can be retrieved for use in scripts. In the following example, the identifiers on the left side of the equal signs are variables:

```
var x:Number = 5;
var name:String = "Lolo";
var c_color:Color = new Color(mcinstanceName);
```

For more information on variables, see "About variables" on page 44.

Syntax

As with all scripting languages, ActionScript has syntax rules that you must follow to create scripts that can compile and run correctly. This section describes the elements that comprise ActionScript syntax:

Case sensitivity

In a case-sensitive programming language, variable names that differ only in case (`book` and `Book`) are considered different from each other. Therefore, it's good practice to follow consistent capitalization conventions, such as those used in this manual, to make it easy to identify names of functions and variables in ActionScript code.

When you publish files for Flash Player 7 or later, Flash implements case sensitivity whether you are using ActionScript 1 or ActionScript 2.0. This means that keywords, class names, variables, method names, and so on are all case sensitive. For example:

```
// In file targeting Flash Player 7
// and either ActionScript 1 or ActionScript 2.0
//
// Sets properties of two different objects
cat.hilite = true;
CAT.hilite = true;

// Creates three different variables
var myVar:Number=10;
var myvar:Number=10;
var mYvAr:Number=10;
// Does not generate an error
var array:Array = new Array();
var date:Date = new Date();
```

This change also affects external variables loaded with `LoadVars.load()`.

Case-sensitivity is implemented for external scripts, such as ActionScript 2.0 class files, scripts that you import using the `#include` command, and scripts in a FLA file. If you encounter runtime errors and are exporting to more than one version of Flash Player, you should review both external script files and scripts in FLA files to confirm that you used consistent capitalization.

Case-sensitivity is implemented on a per-movie basis. If a strict Flash Player 7 application calls a non-strict Flash Player 6 movie, ActionScript executed in the latter movie is non-strict. For example, if you use `loadMovie()` to load a Flash Player 6 SWF into a Flash Player 7 SWF, the version 6 SWF remains case-insensitive, while the version 7 SWF is treated as case-sensitive.

When syntax coloring is enabled, language elements written with correct capitalization are blue by default. For more information, see "Keywords and reserved words" on page 32 and "Syntax highlighting" on page 144.

Dot syntax

In ActionScript, a dot (`.`) is used to access properties or methods belonging to an object or movie clip. It is also used to identify the target path to a movie clip, variable, function, or object. A dot syntax expression begins with the name of the object or movie clip followed by a dot and ends with the element you want to specify.

For example, the `_x` movie clip property indicates a movie clip's *x* axis position on the Stage. The expression `ball_mc._x` refers to the `_x` property of the movie clip instance `ball_mc`.

As another example, `submit` is a variable set in the `form` movie clip, which is nested inside the movie clip `shoppingCart`. The expression `shoppingCart.form.submit = true` sets the `submit` variable of the instance `form` to `true`.

Expressing a method of an object or movie clip follows the same pattern. For example, the `bounce()` method of the `ball` object would be called as follows:

```
ball.bounce();
```

Dot syntax also uses three special aliases, `_root`, `_parent`, and `_global`. The alias `_root` refers to the main Timeline. You can use the `_root` alias to create an absolute target path. For example, the following statement calls the function `buildGameBoard()` in the movie clip `functions` on the main Timeline:

```
_root.functions.buildGameBoard();
```

You can use the alias `_parent` to refer to a movie clip in which the current object is nested. You can also use `_parent` to create a relative target path. For example, if the movie clip `dog_mc` is nested inside the movie clip `animal_mc`, the following statement on the instance `dog_mc` tells `animal_mc` to stop:

```
this._parent.stop();
```

You can use the alias _global to indicate, without having to use a target path, that an object is available to all Timelines in your document. For example, the following statement defines the function `response` that is available to all Timelines:

```
_global.response = function() {
  if (myVar <= 19) {
    myResponse = "25% discount for 20 or more orders";
  } else {
    myResponse = "Thanks for your order";
  }
}
```

For more information, see `_parent`, `_global object`, and `_root`. in *Flash ActionScript Language Reference*.

Slash syntax

Slash syntax was used in Flash 3 and 4 to indicate the target path of a movie clip or variable. This syntax is still supported by Flash Player 7, but its use is not recommended, and slash syntax is not supported in ActionScript 2.0. However, if you are creating content intended specifically for Flash Player 4, you must use slash syntax. For more information, see "Using slash syntax" on page 321.

Curly braces

ActionScript event handlers, class definitions, and functions are grouped together into blocks with curly braces ({}). You can put the opening brace on the same line as your declaration or on the next line, as shown in the following examples. To make your code easier to read, it's a good idea to choose one format and use it consistently. For recommended guidelines on formatting code, see Chapter 3, "Formatting code," on page 76.

The following examples show the opening brace on same line as the declaration:

```
// Event handler
my_btn.onRelease = function() {
  var myDate:Date = new Date();
  var currentMonth:Number = myDate.getMonth();
};

// Class Circle.as
class Circle(radius) {
}

// Function
circleArea = function(radius:Number){
  return radius * radius * Math.PI;
}
```

The following examples show code with opening brace on the next line:

```
//Event handler
my_btn.onRelease = function()
{
  var myDate:Date = new Date();
  var currentMonth:Number = myDate.getMonth();
};

//Class Square.as
class Square(side)
{
}

//Function
squareArea = function(side:Number)
{
  return side * side;
}
```

You can check for matching curly braces in your scripts (see "Checking syntax and punctuation" on page 150).

Semicolons

An ActionScript statement is terminated with a semicolon (;), as shown in the following examples:

```
var column:Number = passedDate.getDay();
var row:Number = 0;
```

If you omit the terminating semicolon, Flash still compiles your script successfully. However, it is good scripting practice to use semicolons because it makes your code more readable.

Semicolons are required within `for` loops, as shown in the following example:

```
//For loop that adds numbers 1-10
var sum:Number = 0;
for (var i=1; i<=10; i++) {
  sum += i;
}
```

Parentheses

When you define a function, place any parameters inside parentheses [()]:

```
function myFunction (name:String, age:Number, reader:Boolean){
  // your code here
}
```

When you call a function, include any parameters passed to the function in parentheses, as shown in the following example:

```
myFunction ("Steve", 10, true);
```

You can also use parentheses to override the ActionScript order of precedence or to make your ActionScript statements easier to read. (See "Operator precedence and associativity" on page 49.)

You also use parentheses to evaluate an expression on the left side of a dot (.) in dot syntax. For example, in the following statement, the parentheses cause `new Color(this)` to evaluate and create a Color object:

```
(new Color(this)).setRGB(0xffffff);
```

If you don't use parentheses, you must add a statement to evaluate the expression, as shown in the following example:

```
myColor = new Color(this);
myColor.setRGB(0xffffff);
```

You can check for matching parentheses in your scripts; see "Checking syntax and punctuation" on page 150. For recommended guidelines on formatting and parentheses, see Chapter 3, "Writing syntax and statements," on page 89.

Comments

Using comments to add notes to scripts is highly recommended. Comments are useful for tracking what you intended and for passing information to other developers if you work in a collaborative environment or are providing samples. Even a simple script is easier to understand if you make notes as you create it.

As shown in the following example, to indicate that a line or portion of a line is a comment, precede the comment with two forward slashes (//):

```
my_btn.onRelease = function() {
  // create new Date object
  var myDate:Date = new Date();
  var currentMonth:Number = myDate.getMonth();
  // convert month number to month name
  var monthName:String = calcMonth(currentMonth);
  var year:Number = myDate.getFullYear();
  var currentDate:Number = myDate.getDate();
};
```

When syntax coloring is enabled (see "Syntax highlighting" on page 144), comments are gray by default. Comments can be any length without affecting the size of the exported file, and they do not need to follow rules for ActionScript syntax or keywords.

To create a comment block, place /* at the beginning of the commented lines and */ at the end. This technique lets you create lengthy comments without adding // at the beginning of each line.

By placing large chunks of script in a comment block, called *commenting out* a portion of your script, you can test specific parts of a script. For example, when the following script runs, none of the code in the comment block is executed:

```
// The following code runs
var x:Number = 15;
var y:Number = 20;
// The following code doesn't run
/*
//create new Date object
var myDate:Date = new Date();
var currentMonth:Number = myDate.getMonth();
// convert month number to month name
var monthName:String = calcMonth(currentMonth);
var year:Number = myDate.getFullYear();
var currentDate:Number = myDate.getDate();}
*/
// The code below runs
var name:String = "My name is";
var age:Number = 20;
```

For recommended guidelines on formatting and parentheses, see Chapter 3, "Using comments in code," on page 77.

Keywords and reserved words

ActionScript reserves words for specific use within the language, so you can't use them as identifiers, such as variable, function, or label names. The following table lists all ActionScript keywords:

add	and	break	case
catch	class	continue	default
delete	do	dynamic	else

eq	extends	finally	for
function	ge	get	gt
if	ifFrameLoaded	implements	import
in	instanceof	interface	intrinsic
le	lt	ne	new
not	on	onClipEvent	or
private	public	return	set
static	switch	tellTarget	this
throw	try	typeof	var
void	while	with	

All class names, component class names, and interface names are reserved words:

Accessibility	Accordion	Alert	Array
Binding	Boolean	Button	Camera
CellRenderer	CheckBox	Collection	Color
ComboBox	ComponentMixins	ContextMenu	ContextMenuItem
CustomActions	CustomFormatter	CustomValidator	DataGrid
DataHolder	DataProvider	DataSet	DataType
Date	DateChooser	DateField	Delta
DeltaItem	DeltaPacket	DepthManager	EndPoint
Error	FocusManager	Form	Function
Iterator	Key	Label	List
Loader	LoadVars	LocalConnection	Log
Math	Media	Menu	MenuBar
Microphone	Mouse	MovieClip	MovieClipLoader
NetConnection	NetStream	Number	NumericStepper
Object	PendingCall	PopUpManager	PrintJob
ProgressBar	RadioButton	RDBMSResolver	Screen
ScrollPane	Selection	SharedObject	Slide
SOAPCall	Sound	Stage	String
StyleManager	System	TextArea	TextField
TextFormat	TextInput	TextSnapshot	TransferObject
Tree	TreeDataProvider	TypedValue	UIComponent
UIEventDispatcher	UIObject	Video	WebService

WebServiceConnector	Window	XML	XMLConnector
XUpdateResolver			

Constants

A constant is a property whose value never changes. ActionScript contains predefined constants.

For example, the constants `BACKSPACE`, `ENTER`, `SPACE`, and `TAB` are properties of the Key object and refer to keyboard keys. To test whether the user is pressing the Enter key, you could use the following statement:

```
if(Key.getCode() == Key.ENTER) {
  alert = "Are you ready to play?";
  control_mc.gotoAndStop(5);
}
```

Flash does not enforce constants; that is, you can't define your own constants.

About data types

A data type describes a piece of data and the kinds of operations that can be performed on it. That data is stored in a variable. You use data types when creating variables, object instances, and function definitions.

ActionScript has the following basic data types that you will probably use frequently in your applications:

- "String data type" on page 35
- "Number data type" on page 36
- "Boolean data type" on page 36
- "Object data type" on page 37
- "MovieClip data type" on page 37
- "Null data type" on page 39
- "Undefined data type" on page 39
- "Void data type" on page 39

ActionScript also has built-in classes, such as Array and Date, that can be considered complex data types. For more information, see "Using the Built-In Classes" on page 197. If you are an advanced developer, you might create custom classes. Any class that you define using the `class` declaration is also considered a data type. All built-in data types and classes are fully defined in *Flash ActionScript Language Reference*.

Variables containing primitive data types behave differently in certain situations than those containing reference types. (See "Using variables in a program" on page 47.) There are also two special data types: null and undefined.

When you debug scripts, you might need to determine the data type of an expression or variable to understand why it is behaving a certain way. You can do this with the `instanceof` operator (see "Determining an item's data type" on page 43).

You can convert one data type to another using one of the following conversion functions: `Array()`, `Boolean()`, `Number()`, `Object()`, `String()`.

In ActionScript 2.0, you can assign data types to variables when you initialize them. The data types you assign can be any of the built-in types or can represent a custom class that you've created. For more information, see "Strict data typing" on page 41.

String data type

A string is a sequence of characters such as letters, numbers, and punctuation marks. You enter strings in an ActionScript statement by enclosing them in single (') or double (") quotation marks.

A common way that you use the string type is to assign a string to a variable. For example, in the following statement, `"L7"` is a string assigned to the variable `favoriteBand_str`:

```
var favoriteBand_str:String = "L7";
```

You can use the addition (+) operator to *concatenate*, or join, two strings. ActionScript treats spaces at the beginning or end of a string as a literal part of the string. The following expression includes a space after the comma:

```
var greeting_str:String = "Welcome, " + firstName;
```

To include a quotation mark in a string, precede it with a backslash character (\). This is called *escaping* a character. There are other characters that cannot be represented in ActionScript except by special escape sequences. The following table provides all the ActionScript escape characters:

Escape sequence	Character
`\b`	Backspace character (ASCII 8)
`\f`	Form-feed character (ASCII 12)
`\n`	Line-feed character (ASCII 10)
`\r`	Carriage return character (ASCII 13)
`\t`	Tab character (ASCII 9)
`\"`	Double quotation mark
`\'`	Single quotation mark
`\\`	Backslash
`\000 - \377`	A byte specified in octal
`\x00 - \xFF`	A byte specified in hexadecimal
`\u0000 - \uFFFF`	A 16-bit Unicode character specified in hexadecimal

Strings in ActionScript are immutable, the same as Java. Any operation that modifies a string returns a new string.

The String class is a built-in ActionScript class. For information on the methods and properties of the String class, see the "String class" entry in *Flash ActionScript Language Reference*.

Number data type

The number data type is a double-precision floating-point number. The minimum value of a number object is approximately 5e-324. The maximum is approximately 1.79E+308.

You can manipulate numbers using the arithmetic operators addition (+), subtraction (-), multiplication (*), division (/), modulo (%), increment (++), and decrement (--). For more information, see "Numeric operators" on page 51.

You can also use methods of the built-in Math and Number classes to manipulate numbers. For more information on the methods and properties of these classes, see the "Math class" and "Number class" entries in *Flash ActionScript Language Reference*.

The following example uses the `sqrt()` (square root) method of the Math class to return the square root of the number 100:

```
Math.sqrt(100);
```

The following example traces a random integer between 10 and 17 (inclusive):

```
var bottles:Number = 0;
bottles = 10 + Math.floor(Math.random()*7);
trace("There are " + bottles + " bottles");
```

The following example finds the percent of the `intro_mc` movie clip loaded and represents it as an integer:

```
var percentLoaded:Number = Math.round((intro_mc.getBytesLoaded()/
  intro_mc.getBytesTotal())*100);
```

Boolean data type

A Boolean value is one that is either `true` or `false`. ActionScript also converts the values `true` and `false` to 1 and 0 when appropriate. Boolean values are most often used with logical operators in ActionScript statements that make comparisons to control the flow of a script.

The following example checks that users enter values into two TextInput component instances. Two Boolean variables are created, `userNameEntered` and `isPasswordCorrect`, and if both variables evaluate to `true`, a welcome message is assigned to the `titleMessage` String variable.

```
//Add two TextInput components and one Button component on the Stage
//Strict data type the three component instances
var userName_ti:mx.controls.TextInput;
var password_ti:mx.controls.TextInput;
var submit_button:mx.controls.Button;

//Create a listener object, which is used with the Button component
//When the Button is clicked, checks for a user name and password
var btnListener:Object = new Object();
btnListener.click = function(evt:Object) {
  //checks that the user enters at least one character in the TextInput
  //instances and returns a Boolean true/false.
  var userNameEntered:Boolean = (userName_ti.text.length>0);
  var isPasswordCorrect:Boolean = (password_ti.text == "vertigo");
  if (userNameEntered && isPasswordCorrect) {
    var titleMessage:String = "Welcome "+userName_ti.text+"!";
```

```
    }
};
submit_button.addEventListener("click", btnListener);
```

For more information, see "Using built-in functions" on page 60 and "Logical operators" on page 53.

Object data type

An object is a collection of properties. Each property has a name and a value. The value of a property can be any Flash data type—even the object data type. This lets you arrange objects inside each other, or *nest* them. To specify objects and their properties, you use the dot (.) operator. For example, in the following code, `hoursWorked` is a property of `weeklyStats`, which is a property of `employee`:

```
employee.weeklyStats.hoursWorked
```

The ActionScript MovieClip object has methods that let you control movie clip symbol instances on the Stage. This example uses the `play()` and `nextFrame()` methods:

```
mcInstanceName.play();
mc2InstanceName.nextFrame();
```

You can also create custom objects to organize information in your Flash application. To add interactivity to an application with ActionScript, you need many pieces of information: for example, you might need a user's name, age, and phone number; the speed of a ball; the names of items in a shopping cart; the number of frames loaded; or the key that was pressed last. Creating custom objects lets you organize this information into groups, simplify your scripting, and reuse your scripts.

The following ActionScript code shows an example of using custom objects to organize information. It creates a new object called `user` and creates three properties: `name`, `age` and `phone` which are String and Numeric data types.

```
var user:Object = new Object();
user.name = "Irving";
user.age = 32;
user.phone = "555-1234";
```

For more information, see "Using classes: a simple example" on page 250.

MovieClip data type

Movie clips are symbols that can play animation in a Flash application. They are the only data type that refers to a graphic element. The MovieClip data type lets you control movie clip symbols using the methods of the MovieClip class.

You do not use a constructor to call the methods of the MovieClip class. You can create a movie clip instance on the Stage or create an instance dynamically. Then you simply call the methods of the MovieClip class using the dot (.) operator.

Working with movie clips on the Stage. The following example calls the `startDrag()` and `getURL()` methods for different movie clip instances that are on the Stage:

```
my_mc.startDrag(true);
parent_mc.getURL("http://www.macromedia.com/support/" + product);
```

The second example returns the width of a movie clip called `my_mc` on the Stage. The targeted instance must be a movie clip, and the returned value must be a numeric value.

```
function getMCWidth(target_mc:MovieClip):Number {
  return target_mc._width;
}
trace(getMCWidth(my_mc));
```

Creating movie clips dynamically Using ActionScript to create movie clips dynamically is useful when you want to avoid manually creating movie clips on the Stage or attaching them from the library. For example, you might create an image gallery with a large number of thumbnails that you want to organize on the Stage. Using `MovieClip.createEmptyMovieClip()` lets you create an application entirely using ActionScript.

To dynamically create a movie clip, use `MovieClip.createEmptyMovieClip()`, as shown in the following example:

```
//Creates a movie clip to hold the container
this.createEmptyMovieClip("image_mc", 9);
//loads an image into image_mc
image_mc.loadMovie("picture.jpg");
```

The second example creates a movie clip called `square_mc` that uses the Drawing API to draw a rectangle. Event handlers and the `startDrag()` and `stopDrag()` methods of the MovieClip class are added to make the rectangle draggable.

```
this.createEmptyMovieClip("square_mc", 1);
square_mc.lineStyle(1, 0x000000, 100);
square_mc.beginFill(0xFF0000, 100);
square_mc.moveTo(100, 100);
square_mc.lineTo(200, 100);
square_mc.lineTo(200, 200);
square_mc.lineTo(100, 200);
square_mc.lineTo(100, 100);
square_mc.endFill();
square_mc.onPress = function() {
  this.startDrag();
};
square_mc.onRelease = function() {
  this.stopDrag();
};
```

For more information, see "Working with Movie Clips" on page 205 and the "MovieClip class" entry in *Flash ActionScript Language Reference*.

Null data type

The null data type has only one value, `null`. This value means *no value*—that is, a lack of data. You can assign the `null` value in a variety of situations to indicate that a property or variable does not yet have a value assigned to it. The following list shows some examples:

- To indicate that a variable exists but has not yet received a value
- To indicate that a variable exists but no longer contains a value
- As the return value of a function, to indicate that no value was available to be returned by the function
- As a parameter to a function, to indicate that a parameter is being omitted

Several methods and functions return `null` if no value has been set. The following example demonstrates how you can use `null` to test if form fields currently have form focus:

```
if (Selection.getFocus() == null) {
  trace("no selection");
}
```

Undefined data type

The undefined data type has one value, `undefined`, and is automatically assigned to a variable to which a value hasn't been assigned, either by your code or user interaction.

The value `undefined` is automatically assigned; unlike `null`, you don't assign `undefined` to a variable or property. You use the undefined data type to check if a variable is set or defined. This data type lets you write code that executes only when the application is running, as shown in the following example:

```
if (init == undefined) {
  trace("initializing app");
  init = true;
}
```

If your application has multiple frames, the code does not execute a second time because the `init` variable is no longer undefined.

Void data type

The void data type has one value, `void`, and is used in a function definition to indicate that the function does not return a value, as shown in the following example:

```
//Creates a function with a return type Void
function displayFromURL(url:String):Void
```

Assigning data types to elements

At runtime, Flash Player automatically assigns data types to the following kinds of language elements, as discussed in the next section:

- Variables
- Parameters passed to a function, method, or class

- Values returned from a function or method
- Objects created as subclasses of existing classes

In ActionScript 2.0, you should explicitly assign data types to items, which can help prevent or diagnose certain errors in your scripts at compile time and offers other benefits. This technique is called *strict data typing*.

For more information on assigning data types, see the following topics:

- "Automatic data typing" on page 40
- "Strict data typing" on page 41
- "Casting objects" on page 42
- "Determining an item's data type" on page 43

Automatic data typing

If you do not explicitly define an item as holding either a number, a string, or another data type, Flash Player will, at runtime, try to determine the data type of an item when it is assigned. If you assign a value to a variable, as shown in the following example, Flash Player evaluates at runtime the element on the right side of the operator and determines that it is of the Number data type:

```
var x = 3;
```

A later assignment might change the type of `x`; for example, the statement `x = "hello"` changes the type of `x` to a string. Because `x` was not declared using strict data typing, the compiler cannot determine the type; to the compiler, the variable `x` can have a value of any type. See "Strict data typing" on page 41.

ActionScript converts data types automatically when an expression requires it and the variables aren't strictly typed. For example, when you pass a value to the `trace()` statement, `trace()` automatically converts the value to a string and sends it to the Output panel.

In expressions with operators, ActionScript converts data types as needed; in the following example, when used with a string, the addition (`+`) operator expects the other operand to be a string:

```
"Next in line, number " + 7
```

ActionScript converts the number 7 to the string `"7"` and adds it to the end of the first string, resulting in the following string:

```
"Next in line, number 7"
```

Strict data typing is recommended; for more information, see "Strict data typing" on page 41.

Strict data typing

ActionScript 2.0 lets you explicitly declare the object type of a variable when you create it, which is called *strict data typing*. Strict data typing offers several benefits at compile time. Because data type mismatches trigger compiler errors, strict data typing helps you find bugs in your code at compile time and prevents you from assigning the wrong type of data to an existing variable. During authoring, strict data typing activates code hinting in the ActionScript editor (but you should still use instance-name suffixes for visual elements). Although strict data typing is relevant only at compile time, it can increase performance at runtime by making your scripts run faster.

To assign a specific data type to an item, specify its type using the `var` keyword and post-colon syntax, as shown in the following example:

```
// strict typing of variable or object
var x:Number = 7;
var birthday:Date = new Date();

// strict typing of parameters
function welcome(firstName:String, age:Number){
}

// strict typing of parameter and return value
function square(x:Number):Number {
  var squared:Number = x*x;
  return squared;
}
```

Because you must use the `var` keyword when strictly typing variable, you can't strictly type a global variable (see "Scoping and declaring variables" on page 45).

You can declare the data type of objects based on built-in classes (Button, Date, MovieClip, and so on) and on classes and interfaces that you create. In the following example, if you have a file named Student.as in which you define the Student class, you can specify that objects you create are of type Student:

```
var student:Student = new Student();
```

You can also specify that objects are of type Function or Void.

Using strict data typing helps ensure that you don't inadvertently assign an incorrect type of value to an object. Flash checks for typing mismatch errors at compile time. For example, suppose you type the following code:

```
// in the Student.as class file
class Student {
  var status:Boolean; // property of Student objects
}

// in a script
var studentMaryLago:Student = new Student();
studentMaryLago.status = "enrolled";
```

When Flash compiles this script, a "Type mismatch" error is generated because the SWF is expecting a Boolean value.

Using strict typing also helps to ensure that you do not attempt to access properties or methods that are not part of an object's type.

Another advantage of strict data typing is that Flash MX 2004 automatically shows code hints for built-in objects when they are strictly typed. For more information, see "Strictly typing objects to trigger code hints" on page 145.

Files published using ActionScript 1 do not respect strict data typing assignments at compile time, so assigning the wrong type of value to a variable that you have strictly typed doesn't generate a compiler error.

```
var x:String = "abc"
x = 12 ; // no error in ActionScript 1, type mismatch error in ActionScript 2
```

The reason for this is that when you publish a file for ActionScript 1, Flash interprets a statement such as `var x:String = "abc"` as slash syntax rather than as strict typing. (ActionScript 2.0 doesn't support slash syntax.) This behavior can result in an object that is assigned to a variable of the wrong type, causing the compiler to let illegal method calls and undefined property references pass through unreported.

Therefore, if you are implementing strict data typing, make sure you are publishing files for ActionScript 2.0.

Casting objects

ActionScript 2.0 lets you cast one data type to another. The cast operator that Flash uses takes the form of a function call and is concurrent with *explicit coercion*, as specified in the ECMA-262 Edition 4 proposal (see www.mozilla.org/js/language/es4/index.html). Casting lets you assert that an object is of a certain type so that when type-checking occurs, the compiler treats the object as having a set of properties that its initial type does not contain. This can be useful, for example, when iterating over an array of objects that might be of differing types but share a base type.

The syntax for casting is *`type(item)`*, where you want the compiler to behave as if the data type of *`item`* is *`type`*. Casting is essentially a function call, and the function call returns `null` if the cast fails at runtime (in files published for Flash Player 7 or later; files published for Flash Player 6 do not have runtime support for failed casts). If the cast succeeds, the function call returns the original object. However, the compiler cannot determine whether a cast will fail at runtime and won't generate compile-time errors in those cases. The following code shows an example:

```
function bark(myAnimal:Animal) {
var foo:Dog = Dog(myAnimal);
foo.bark();
}

var curAnimal:Animal = new Dog();
bark(curAnimal); // will work
curAnimal = new Cat();
bark(curAnimal); // won't work
```

In this situation, you asserted to the compiler that `foo` is a Dog object, and, therefore, the compiler assumes that `temp.bark();` is a legal statement. However, the compiler doesn't know that the cast will fail (that is, that you tried to cast a Cat object to an Animal type), so no compile-time error occurs. If you include a check in your script to make sure that the cast succeeds, you can find casting errors at runtime.

```
import Dog;
function bark(myAnimal:Animal) {
  var foo:Dog = Dog(myAnimal);
  if (foo) {
    foo.bark();
  }
}
```

You can cast an expression to an interface. If the expression is an object that implements the interface or has a base class that implements the interface, the cast succeeds. If not, the cast fails.

Casting to null or undefined returns `undefined`.

You can't override primitive data types that have a corresponding global conversion function with a cast operator of the same name. This is because the global conversion functions have precedence over the cast operators. For example, you can't cast to Array because the `Array()` conversion function takes precedence over the cast operator. For more information on data conversion functions, see the entry for each conversion function in *Flash ActionScript Language Reference*: `Array()`, `Boolean()`, `Number()`, `Object()`, `String()`.

Determining an item's data type

While testing and debugging your programs, you might discover problems that seem to be related to the data types of different items. In these cases, you may want to determine an item's data type. You can use either the `typeof` operator or the `instanceof` operator.

Use the `typeof` operator to get the data types; `typeof` does not return information about which class to which an instance belongs. Use the `instanceof` operator to determine if an object is of a specified data type or not; `instanceof` returns a Boolean value.

The following example shows how you can use these operators and the difference between them:

```
//Create a new instance of LoadVars class
var myLV:LoadVars = new LoadVars();

//instanceof operator specifies instance of what class
if (myLV instanceof LoadVars) {
  trace("yes, it's a loadvars instance");
}

//typeof operator does not specify class, only specifies that myLV is an object
var typeResult:String = typeof(myLV);
trace(typeResult);
```

For more information about these operators, see `typeof` and `instanceof` in *Flash ActionScript Language Reference*. For more information on testing and debugging, see Chapter 4, "Writing and Debugging Scripts," on page 139.

About variables

A *variable* is a container that holds information. The container itself is always the same, but the contents can change. By changing the value of a variable as the SWF file plays, you can record and save information about what the user has done, record values that change as the SWF file plays, or evaluate whether a condition is true or false.

It's a good idea always to assign a variable a known value the first time you define the variable. This is known as *initializing a variable* and is often done in the first frame of the SWF file. If a variable is declared in the FLA (instead of in an external file), it works only for the frame where the variable is declared. Only the frame code for that frame is scanned, so the second frame would not recognize the variable if it was referenced. Initializing a variable helps you track and compare the variable's value as the SWF file plays. Flash Player 7 and later evaluate uninitialized variables differently than Flash Player 6 and earlier. If you have written scripts for Flash Player 6 and plan to write or port scripts for Flash Player 7 or later, you should be understand these differences to avoid unexpected behavior. For more information, see "ECMA-262 compliance" on page 14.

Variables can hold any type of data (see "About data types" on page 34). The type of data a variable contains affects how the variable's value changes when it is assigned in a script.

Typical types of information that you can store in a variable include a URL, a user's name, the result of a mathematical operation, the number of times an event occurred, or whether a button has been clicked. Each SWF file and movie clip instance has a set of variables, with each variable having a value independent of variables in other SWF files or movie clips.

To view the value of a variable, use the `trace()` statement to send the value to the Output panel. For example, `trace(hoursWorked)` sends the value of the variable `hoursWorked` to the Output panel in test mode. You can also check and set the variable values in the Debugger in test mode. For more information, see "Using the trace statement" on page 165 and "Displaying and modifying variables" on page 157.

For more information, see the following topics:

- "Naming a variable" on page 44
- "Scoping and declaring variables" on page 45
- "Using variables in a program" on page 47

Naming a variable

A variable's name must follow these rules:

- It must be an identifier (see "Terminology" on page 24).
- It cannot be a keyword or an ActionScript literal such as `true`, `false`, `null`, or `undefined`. (See Chapter 3, "Avoiding reserved words," on page 75).
- It must be unique within its scope (see "Scoping and declaring variables" on page 45).

You should not use any element in the ActionScript language as a variable name because it can cause syntax errors or unexpected results. In the following example, if you name a variable `String` and then try to create a String object using `new String()`, the new object is undefined:

```
// This code works as expected
var hello_str:String = new String();
trace(hello_str.length); // returns 0

// But if you give a variable the same name as a built-in class....
var String:String = "hello";
var hello_str:String = new String();
trace(hello_str.length); // returns undefined
```

The ActionScript editor supports code hints for built-in classes and for variables that are based on these classes. If you want Flash to provide code hints for a particular object type that is assigned to a variable, you can strictly type the variable.

For example, suppose you type the following code:

```
var members:Array = new Array();
members.
```

As soon as you type the period (.), Flash displays a list of methods and properties available for Array objects.

Another way to tell Flash to provide code hints is to name the variable using a specific suffix. For more information on using strict typing and suffixes, see "Writing code that triggers code hints" on page 145. For information and detailed guidelines on naming variables, see Chapter 3, "General naming guidelines," on page 69.

Scoping and declaring variables

A variable's *scope* refers to the area in which the variable is known and can be referenced. There are three types of variable scopes in ActionScript:

- Timeline variables are available to any script on that Timeline.
- Local variables are available within the function body in which they are declared (delineated by curly braces).
- Global variables and functions are visible to every Timeline and scope in your document.

For guidelines on using scope and variables, see Chapter 3, "Using scope," on page 95 and Chapter 3, "Avoiding _root," on page 96.

Note: ActionScript 2.0 classes that you create support public, private, and static variable scopes. For more information, see "Controlling member access" on page 256 and "Creating class members" on page 263.

Timeline variables

Timeline variables are available to any script on that Timeline. To declare Timeline variables, use the `var` statement and initialize them in any frame in the Timeline; the variable will be available to that frame and all following frames, as shown in the following example:

```
var x:Number = 15; //initialized in Frame 1, so it's available to all frames
```

Make sure to declare a Timeline variable before trying to access it in a script. For example, if you put the code `var x = 10;` in Frame 20, a script attached to any frame before Frame 20 cannot access that variable.

Local variables

To declare local variables, use the `var` statement inside the body of a function. A local variable declared within a function block is defined within the scope of the function block and expires at the end of the function block.

For example, the variables `i` and `j` are often used as loop counters. In the following example, `i` is used as a local variable; it exists only inside the function `initArray()`:

```
var myArray:Array = [ ];
function initArray(arrayLength:Number) {
  var i:Number;
  for( i = 0; i < arrayLength; i++ ) {
    myArray[i] = i + 1;

  }
}
```

Local variables can also help prevent name conflicts, which can cause errors in your application. For example, if you use `age` as a local variable, you could use it to store a person's age in one context and the age of a person's child in another; because these variables would run in separate scopes, there would be no conflict.

It's good practice to use local variables in the body of a function so that the function can act as an independent piece of code. A local variable is changeable only within its own block of code. If an expression in a function uses a global variable, something outside the function can change its value, which would change the function.

You can assign a data type to a local variable when you define it, which helps prevent you from assigning the wrong type of data to an existing variable. For more information, see "Strict data typing" on page 41.

Global variables

Global variables and functions are visible to every Timeline and scope in your document. To create a variable with global scope, use the `_global` identifier before the variable name and do not use the `var =` syntax. For example, the following code creates the global variable `myName`:

```
var _global.myName = "George"; // incorrect syntax for global variable
_global.myName = "George"; // correct syntax for global variable
```

However, if you initialize a local variable with the same name as a global variable, you don't have access to the global variable while you are in the scope of the local variable, as shown in the following example:

```
_global.counter = 100; // declares global variable
trace(counter); // accesses the global variable and displays 100
function count(){
   for( var counter = 0; counter <= 2 ; counter++ ) { //local variable
   trace(counter); // accesses local variable and displays 0 through 2
```

```
    }
}
count();
trace(counter); // accesses global variable and displays 100
```

This example simply shows that the global variable is not accessed in the scope of the `count()` function. To avoid confusion in your applications, name your variables uniquely.

The Flash Player version 7 and later security sandbox enforces restrictions when accessing global variables from movies loaded from separate security domains. For more information, see "Flash Player security features" on page 288.

Using variables in a program

You must declare and initialize a variable in a script before you can use it in an expression. If you use an undeclared variable, as shown in the following example, the variable's value in Flash Player 7 will be `NaN` or `undefined`, and your script might produce unintended results:

```
var squared:Number = x*x;
trace(squared); // NaN in Flash Player 7; 0 in earlier versions
var x:Number = 6;
```

In the following example, the statement declaring and initializing the variable `x` comes first, so `squared` can be replaced with a value:

```
var x:Number = 6;
var squared:Number = x*x;
trace(squared); // 36
```

Similar behavior occurs when you pass an undefined variable to a method or function:

```
//does not work
getURL(myWebSite); // no action
var myWebSite = "http://www.macromedia.com";
//works
var myWebSite = "http://www.macromedia.com";
getURL(myWebSite); // browser displays www.macromedia.com
```

You can change the value of a variable in a script as many times as you want.

The type of data that a variable contains affects how and when the variable's value changes. Primitive data types, such as strings and numbers, are "passed by value"; this means that the current value of the variable is used, rather than a reference to that value.

In the following example, `x` is set to 15 and that value is copied into `y`. When `x` is changed to 30 in line 3, the value of `y` remains 15, because `y` doesn't look to `x` for its value; it contains the value of `x` that it received in line 2.

```
var x:Number = 15;
var y:Number = x;
var x:Number = 30;
trace(x);  // 30
trace(y);  // 15
```

In the following example, the variable `inValue` contains a primitive value, 3, so that value is passed to the `sqr()` function and the returned value is 9:

```
function sqr(x:Number):Number {
  var x:Number = x * x;
  return x;
}

var inValue:Number = 3;
var out:Number = sqr(inValue);
trace(inValue); //3
trace(out);     //9
```

The value of the variable `inValue` does not change, even though the value of `x` in the function changes.

The object data type can contain such a large amount of complex information that a variable with this type doesn't hold an actual value; it holds a reference to a value. This reference is similar to an alias that points to the contents of the variable. When the variable needs to know its value, the reference asks for the contents and returns the answer without transferring the value to the variable.

The following example shows passing by reference:

```
var myArray:Array = ["tom", "josie"];
var newArray:Array = myArray;
myArray[1] = "jack";
trace(newArray); // tom,jack
```

This code creates an Array object called `myArray` that has two elements. The variable `newArray` is created and is passed a reference to `myArray`. When the second element of `myArray` is changed to `"jack"`, it affects every variable with a reference to it. The `trace()` statement sends `tom,jack` to the Output panel. Flash uses a zero-based index, which means that 0 is the first item in the array, 1 is the second, and so on.

In the following example, `myArray` contains an Array object, so it is passed to function `zeroArray()` by reference. The function `zeroArray()` accepts an Array object as a parameter and sets all the elements of that array to 0. It can modify the array because the array is passed by reference.

```
function zeroArray (theArray:Array):Void {
  var i:Number;
  for (i=0; i < theArray.length; i++) {
    theArray[i] = 0;
  }
}

var myArray:Array = new Array();
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
zeroArray(myArray);
trace(myArray); // 0,0,0
```

Using operators to manipulate values in expressions

An expression is any statement that Flash can evaluate and that returns a value. You can create an expression by combining operators and values or by calling a function.

Operators are characters that specify how to combine, compare, or modify the values of an expression. The elements that the operator performs on are called *operands*. For example, in the following statement, the addition (`+`) operator adds the value of a numeric literal to the value of the variable `foo`; `foo` and `3` are the operands:

```
foo + 3
```

This section describes general rules about common types of operators, operator precedence, and operator associativity:

- "Operator precedence and associativity" on page 49
- "Numeric operators" on page 51
- "Comparison operators" on page 52
- "String operators" on page 53
- "Logical operators" on page 53
- "Bitwise operators" on page 54
- "Equality operators" on page 54
- "Assignment operators" on page 55
- "Dot and array access operators" on page 56

For information on these operators, as well as special operators that don't fall into these categories, see Chapter 2, "ActionScript Language Reference" in *Flash ActionScript Language Reference*.

Operator precedence and associativity

When two or more operators are used in the same statement, some operators take precedence over others. ActionScript follows a precise hierarchy to determine which operators to execute first. For example, multiplication is always performed before addition; however, items in parentheses [()] take precedence over multiplication. So, without parentheses, ActionScript performs the multiplication in the following example first:

```
total = 2 + 4 * 3;
```

The result is 14.

But when parentheses surround the addition operation, ActionScript performs the addition first:

```
total = (2 + 4) * 3;
```

The result is 18.

When two or more operators share the same precedence, their associativity determines the order in which they are performed. Associativity can be either left-to-right or right-to-left. For example, the multiplication (*) operator has an associativity of left-to-right; therefore, the following two statements are equivalent:

```
total = 2 * 3 * 4;
total = (2 * 3) * 4;
```

The following table lists all the ActionScript operators and their associativity, from highest to lowest precedence. Deprecated Flash 4 operators are listed in Appendix B, "Deprecated Flash 4 operators," on page 311.For more information and guidelines on using operators and parentheses, see Chapter 3, "Formatting code," on page 76.

Operator	Description	Associativity
	Highest precedence	
x++	Post-increment	Left to right
x--	Post-decrement	Left to right
.	Object property access	Left to right
[ ]	Array element	Left to right
( )	Parentheses	Left to right
function ( )	Function call	Left to right
++*x*	Pre-increment	Right to left
--*x*	Pre-decrement	Right to left
-	Unary negation, such as x = -1	Left to right
~	Bitwise NOT	Right to left
!	Logical NOT	Right to left
new	Allocate object	Right to left
delete	Deallocate object	Right to left
typeof	Type of object	Right to left
void	Returns undefined value	Right to left
*	Multiply	Left to right
/	Divide	Left to right
%	Modulo	Left to right
+	Unary plus	Right to left
-	Unary minus	Right to left
<<	Bitwise left shift	Left to right
>>	Bitwise right shift	Left to right
>>>	Bitwise right shift (unsigned)	Left to right

Operator	Description	Associativity
instanceof	Instance of (finds the class of which the object is an instance) Requires Flash Player 6 or later	Left to right
<	Less than	Left to right
<=	Less than or equal to	Left to right
>	Greater than	Left to right
>=	Greater than or equal to	Left to right
==	Equal	Left to right
!=	Not equal	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional	Right to left
=	Assignment	Right to left
*=, /=, %=, +=, -=, &=, |=, ^=, <<=, >>=, >>>=	Compound assignment	Right to left
,	Comma	Left to right
	Lowest precedence	

Numeric operators

Numeric operators add, subtract, multiply, divide, and perform other arithmetic operations.

The most common use of the increment operator is `i++` instead of the more verbose `i = i+1`. You can use the increment operator before or after an operand. In the following example, `age` is incremented first and then tested against the number 30:

```
if (++age >= 30)
```

This process is also known as a preincrement. In the following example, `age` is incremented after the test is performed:

```
if (age++ >= 30)
```

This process is also known as a postincrement. The following table lists the ActionScript numeric operators:

Operator	Operation performed
+	Addition
*	Multiplication

Operator	Operation performed
/	Division
%	Modulo (remainder of division)
-	Subtraction
++	Increment
--	Decrement

Comparison operators

Comparison operators compare the values of expressions and return a Boolean value (`true` or `false`). These operators are most commonly used in loops and in conditional statements. In the following example, if the variable `score` is `100`, a certain function is called; otherwise, a different function is called:

```
// call one function or another based on score
if (score > 100){
  highScore();
}
else {
  lowScore();
}
```

In the following example, if the user's entry (a string variable, `userEntry`) matches their stored password, the playhead moves to a named frame called `welcomeUser`:

```
if (userEntry == userPassword) {
  gotoAndStop("welcomeUser");
}
```

Except for the strict equality (==) operator, the comparison operators compare strings only if both operands are strings. If only one of the operands is a string, ActionScript converts both operands to numbers and performs a numeric comparison. Uppercase characters precede lowercase in alphabetic order, so "Eagle" comes before "dog." If you want to compare two strings or characters regardless of case, you need to convert both strings to upper- or lowercase before comparing them.

The following table lists the ActionScript comparison operators:

Operator	Operation performed
<	Less than: Returns `true` if the left operand is mathematically smaller than the right operand. Returns `true` if the left operand alphabetically precedes the right operand (for example, a < b).
>	Greater than: Returns `true` if the left operand is mathematically larger than the right operand. Returns `true` if the left operand alphabetically follows the right operand (for example, b > a).

Operator	Operation performed
<=	Less than or equal to: Returns `true` if the left operand is mathematically smaller than or the same as the right operand. Returns `true` if the left operand alphabetically precedes or is the same as the right operand.
>=	Greater than or equal to: Returns `true` if the left operand is mathematically larger than or the same as the right operand. Returns `true` if the left operand alphabetically follows or is the same as the right operand.
<> !=	Not equal: Returns `true` if the operands are not mathematically equivalent. Returns `true` if the operands are not the same.
==	Equality: Tests two expressions for equality. The result is `true` if the expressions are equal.
===	Strict equality: Tests two expressions for equality; the strict equality operator performs the same as the equality operator except that data types are not converted. The result is `true` if both expressions, including their data types, are equal. Does not apply to strings.

String operators

The addition (+) operator has a special effect when it operates on strings: It concatenates the two string operands. For example, the following statement adds `"Congratulations, "` to `"Donna!"`:

```
"Congratulations, " + "Donna!"
```

The result is `"Congratulations, Donna!"` If only one of the addition (+) operator's operands is a string, Flash converts the other operand to a string. ActionScript treats spaces at the beginning or end of a string as a literal part of the string.

The comparison operators >, >=, <, and <= also have a special effect when operating on strings. These operators compare two strings to determine which is first in alphabetical order.
The comparison operators compare strings only if both operands are strings. If only one of the operands is a string, ActionScript converts both operands to numbers and performs a numeric comparison. Uppercase characters precede lowercase in alphabetic order, so "Eagle" comes before "dog." If you want to compare two strings or characters regardless of case, you need to convert both strings to upper- or lowercase before comparing them.

Logical operators

Logical operators compare Boolean values (`true` and `false`) and return a third Boolean value. For example, if both operands evaluate to `true`, the logical AND (`&&`) operator returns `true`. If one or both of the operands evaluate to `true`, the logical OR (`||`) operator returns `true`. Logical operators are often used with comparison operators to determine the condition of an `if` statement. For example, in the following script, if both expressions are true, the `if` statement will execute and the `myFunc()` function will be called:

```
if (i > 10 && i <50){
  myFunc(i);
}
```

Consider the order of operands, especially if you're setting up complex conditions and you know how often one condition is true compared with other conditions. In the previous example, if you know that, in most cases, `i` will be greater than 50, consider putting the condition `i<50` first; the condition `i<50` will be checked first, and the second condition doesn't need to be checked in most cases.

The following table lists the ActionScript logical operators:

Operator	Operation performed
`&&`	Logical AND: Returns `true` only if both the left and right operands are true.
`||`	Logical OR: Returns `true` if either the left or right operand is true.
`!`*operand*	Logical NOT: Returns the logical (Boolean) opposite of the operand. The logical NOT operator takes one operand.

Bitwise operators

Bitwise operators internally manipulate floating-point numbers to change them into 32-bit integers. The exact operation performed depends on the operator, but all bitwise operations evaluate each binary digit (bit) of the 32-bit integer individually to compute a new value.

The following table lists the ActionScript bitwise operators:

Operator	Operation performed
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
«	Shift left
»	Shift right
»»	Shift right zero fill

Equality operators

You can use the equality (==) operator to determine whether the values or references of two operands are equal. This comparison returns a Boolean (`true` or `false`) value. If the operands are strings, numbers, or Boolean values, they are compared by value. If the operands are objects or arrays, they are compared by reference.

It is a common mistake to use the assignment operator to check for equality. For example, the following code compares *x* to 2:

```
if (x == 2)
```

In that same example, the expression `if (x = 2)` would be incorrect, because it doesn't compare the operands; it assigns the value of 2 to the variable `x`.

The strict equality (===) operator is similar to the equality operator, with one important difference: The strict equality operator does not perform type conversion. If the two operands are of different types, the strict equality operator returns `false`. The strict inequality (`!==`) operator returns the opposite of the strict equality operator.

The following table lists the ActionScript equality operators:

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

Assignment operators

You can use the assignment (`=`) operator to assign a value to a variable, as shown in the following example:

```
var password:String = "Sk8tEr";
```

You can also use the assignment operator to assign multiple variables in the same expression. In the following statement, the value of `d` is assigned to the variables `a`, `b`, and `c`:

```
a = b = c = d;
```

You can also use compound assignment operators to combine operations. Compound operators perform on both operands and then assign the new value to the first operand. For example, the following two statements are equivalent:

```
x += 15;
x = x + 15;
```

The assignment operator can also be used in the middle of an expression, as shown in the following example:

```
// If the flavor is not vanilla, output a message.
if ((flavor = getIceCreamFlavor()) != "vanilla") {
  trace ("Flavor was " + flavor + ", not vanilla.");
}
```

This code is equivalent to the following code, which is slightly easier to read:

```
flavor = getIceCreamFlavor();
if (flavor != "vanilla") {
  trace ("Flavor was " + flavor + ", not vanilla.");
}
```

The following table lists the ActionScript assignment operators:

Operator	Operation performed
`=`	Assignment
`+=`	Addition and assignment

Operator	Operation performed
-=	Subtraction and assignment
*=	Multiplication and assignment
%=	Modulo and assignment
/=	Division and assignment
<<=	Bitwise shift left and assignment
>>=	Bitwise shift right and assignment
>>>=	Shift right zero fill and assignment
^=	Bitwise XOR and assignment
|=	Bitwise OR and assignment
&=	Bitwise AND and assignment

Dot and array access operators

You can use the dot operator (`.`) and the array access operator (`[]`) to access built-in or custom ActionScript object properties, including those of a movie clip.

Dot operator. The dot operator uses the name of an object on its left side and the name of a property or variable on its right side. The property or variable name can't be a string or a variable that evaluates to a string; it must be an identifier. The following examples use the dot operator:

```
year.month = "June";
year.month.day = 9;
```

The dot operator and the array access operator perform the same role, but the dot operator takes an identifier as its property, whereas the array access operator evaluates its contents to a name and then accesses the value of that named property.

For example, the following expressions access the same variable `velocity` in the movie clip `rocket`:

```
rocket.velocity;
rocket["velocity"];
```

Array access operator. You can use the array access operator to dynamically set and retrieve instance names and variables. For example, in the following code, the expression inside the array access operator is evaluated, and the result of the evaluation is used as the name of the variable to be retrieved from movie clip `name`:

```
name["mc" + i]
```

In ActionScript 2.0, you can use the bracket operator to access dynamically created properties, if the dynamic keyword was not used to define the class, as shown in the following example:

```
class Foo {
}

// in a fla that uses Foo
var bar:Foo = new Foo();
```

```
function barGetProp():String {
return "bar";
}
function barSetProp(str:String):Void {
}
bar.addProperty("someProp", barGetProp, barSetProp);
//trace(bar.someProp); // generates an error
trace(bar["someProp"]); // no error
```

You can use the `eval()` function to dynamically set and retrieve instance names and variables, as shown in the following example:

```
eval("mc" + i)
```

The array access operator can also be used on the left side of an assignment statement. This lets you dynamically set instance, variable, and object names, as shown in the following example:

```
name[index] = "Gary";
```

You create multidimensional arrays in ActionScript by constructing an array, the elements of which are also arrays. To access elements of a multidimensional array, you can nest the array access operator with itself, as shown in the following example:

```
var chessboard:Array = new Array();
for (var i=0; i<8; i++) {
  chessboard.push(new Array(8));
}
function getContentsOfSquare(row, column){
  chessboard[row][column];
}
```

When you use the array access operator, the ActionScript compiler cannot check if the accessed element is a valid property of the object.

You can check for matching `[]` operators in your scripts; see "Checking syntax and punctuation" on page 150.

Specifying an object's path

To use an action to control a movie clip or loaded SWF file, you must specify its name and its address, called a *target path*.

In ActionScript, you identify a movie clip by its instance name, either in the Property inspector or dynamically, at runtime. For example, in the following statement, the `_alpha` property of the movie clip named `star` is set to 50% visibility:

```
star._alpha = 50;
```

To give a movie clip on the Stage an instance name:

1. Select the movie clip on the Stage.
2. Enter an instance name in the Property inspector in the Instance Name text box.

To dynamically create a movie clip:

- Use the `MovieClip.attachMovie()`, `MovieClip.createEmptyMovieClip()`, or `MovieClip.duplicateMovieClip()` method. The following example uses the `attachMovie()` method to dynamically create the `purpleDot2_mc` movie clip and place it on top of the `greenBox_mc` movie clip when the user rolls over the `greenBox_mc` clip, which is on the Stage:

```
my_btn.onRollOut = function(){
  this.greenBox_mc.attachMovie("purpleDot", "purpleDot2_mc", 2);
}
```

To identify a loaded SWF file:

- Use `_levelX`, where *X* is the level number specified in the `loadMovie()` function that loaded the SWF file.

  For example, a SWF file loaded into level 5 has the target path `_level5`. In the following example, a SWF file is loaded into level 5 and its visibility is set to `false`:

```
//Load the SWF onto level 99.
loadMovieNum("contents.swf", 99);
//Set the visibility of level 99 to false.
loader_mc.onEnterFrame = function(){
  _level99._visible = false;
};
```

To enter a SWF file's target path:

- In the Actions panel (Window > Development > Actions), click the Insert Target Path button
- Select a movie clip from the list that appears.

For more information on target paths, see "Using absolute and relative target paths" in *Using Flash*.

Using condition statements

To perform an action depending on whether a condition exists, or to repeat an action (create loop statements), you can use `if`, `else`, `else if`, `for`, `while`, `do while`, `for..in`, or `switch` statements.

Checking conditions

Statements that check whether a condition is `true` or `false` begin with the term `if`. If the condition evaluates to `true`, ActionScript executes the next statement. If the condition doesn't exist, ActionScript skips to the next statement outside the block of code.

To optimize your code's performance, check for the most likely conditions first.

The following statements test three conditions. The term `else if` specifies alternative tests to perform if previous conditions are false.

```
if (password == null || email == null) {
  gotoAndStop("reject");
} else if (password == userID){
```

```
  gotoAndPlay("startProgram");
}
```

If you want to check for one of several conditions, you can use the `switch` statement rather than multiple `else if` statements.

Repeating actions

ActionScript can repeat an action a specified number of times or while a specific condition exists. Use the `while`, `do..while`, `for`, and `for..in` actions to create loops.

To repeat an action while a condition exists:

- Use the `while` statement.

A `while` loop evaluates an expression and executes the code in the body of the loop if the expression is `true`. After each statement in the body is executed, the expression is evaluated again. In the following example, the loop executes four times:

```
i = 4;
while (i>0) {
  my_mc.duplicateMovieClip("newMC"+i, i, {_x:i*20, _y:i*20});
  i--;
}
```

You can use the `do..while` statement to create the same kind of loop as a `while` loop. In a `do..while` loop, the expression is evaluated at the bottom of the code block so the loop always runs at least once.

This is shown in the following example:

```
i = 4;
do {
  my_mc.duplicateMovieClip("newMC"+i, i, {_x:i*20, _y:i*20});
  i--;
} while (i>0);
```

To repeat an action using a built-in counter:

- Use the `for` statement.

Most loops use some kind of counter to control how many times the loop executes. Each execution of a loop is called an *iteration*. You can declare a variable and write a statement that increases or decreases the variable each time the loop executes. In the `for` action, the counter and the statement that increments the counter are part of the action. In the following example, the first expression (`var i = 4`) is the initial expression that is evaluated before the first iteration. The second expression (`i > 0`) is the condition that is checked each time before the loop runs. The third expression (`i--`) is called the *post expression* and is evaluated each time after the loop runs.

```
for (var i = 4; i > 0; i--){
  my_mc.duplicateMovieClip("newMC"+ i, i, {_x:i*20, _y:i*20});
}
```

To loop through the children of a movie clip or an object:

- Use the `for..in` statement.

Children include other movie clips, functions, objects, and variables. The following example uses the `trace` statement to print its results in the Output panel:

```
var myObject:Object = { name:'Joe', age:25, city:'San Francisco' };
for (propertyName in myObject) {
  trace("myObject has the property: " + propertyName + ", with the value: " +
  myObject[propertyName]);
}
```

This example produces the following results in the Output panel:

```
myObject has the property: name, with the value: Joe
myObject has the property: age, with the value: 25
myObject has the property: city, with the value: San Francisco
```

You might want your script to iterate over a particular type of child—for example, over only movie clip children. You can do this using `for..in` with the `typeof` operator.

```
for (myname in my_object) {
  if (typeof (my_object[myname]) == "anObject") {
    trace("I have an object child named " + myname);
  }
}
```

For more information on each action, see the individual entries in *Flash ActionScript Language Reference*: `while`, `do while`, `for`, `for..in`.

Using built-in functions

A function is a block of ActionScript code that can be reused anywhere in a SWF file. If you pass values as parameters to a function, the function will operate on those values. A function can also return values.

Flash has built-in functions that let you access certain information and perform certain tasks, such as getting the version number of Flash Player that is hosting the SWF file (`getVersion()`). Functions that belong to an object are called *methods*. Functions that don't belong to an object are called *top-level functions* and are found in the Functions category of the Actions panel.

Each function has individual characteristics, and some functions require you to pass certain values. If you pass more parameters than the function requires, the extra values are ignored. If you don't pass a required parameter, the empty parameters are assigned the `undefined` data type, which can cause errors during runtime. To call a function, it must be in a frame that the playhead has reached.

To call a function, simply use the function name and pass any required parameters:

```
isNaN(someVar);
getTimer();
eval("someVar");
```

For more information on each function, see its entry in *Flash ActionScript Language Reference*.

Creating functions

You can define functions to execute a series of statements on passed values. Your functions can also return values. After a function is defined, it can be called from any Timeline, including the Timeline of a loaded SWF file.

A well-written function can be thought of as a "black box." If it has carefully placed comments about its input, output, and purpose, a user of the function does not need to understand exactly how the function works internally.

For more information, see the following topics:

Defining a function

As with variables, functions are attached to the Timeline of the movie clip that defines them, and you must use a target path to call them. As with variables, you can use the `_global` identifier to declare a global function that is available to all Timelines and scopes without using a target path. To define a global function, precede the function name with the identifier `_global`, as shown in the following example:

```
_global.myFunction = function (x:Number):Number {
   return (x*2)+3;
}
```

To define a Timeline function, use the `function` statement followed by the name of the function, any parameters to be passed to the function, and the ActionScript statements that indicate what the function does.

The following example is a function named `areaOfCircle` with the parameter `radius`:

```
function areaOfCircle(radius:Number):Number {
  return Math.PI * radius * radius;
}
```

You can also define a function by creating a *function literal*—an unnamed function that is declared in an expression instead of in a statement. You can use a function literal to define a function, return its value, and assign it to a variable in one expression, as shown in the following example:

```
area = (function() {return Math.PI * radius *radius;})(5);
```

When a function is redefined, the new definition replaces the old definition.

For information on strictly typing function return types and parameters, see "Strict data typing" on page 41.

Passing parameters to a function

Parameters are the elements on which a function executes its code. (In this manual, the terms *parameter* and *argument* are interchangeable.) For example, the following function takes the parameters `initials` and `finalScore`:

```
function fillOutScorecard(initials:String, finalScore:Number):Void {
  scorecard.display = initials;
  scorecard.score = finalScore;
}
```

When the function is called, the required parameters must be passed to the function. The function substitutes the passed values for the parameters in the function definition. In this example, `scorecard` is the instance name of an object; `display` and `score` are properties of the object. The following function call assigns the value `"JEB"` to the variable `display` and the value `45000` to the variable `score`:

```
fillOutScorecard("JEB", 45000);
```

The parameter `initials` in the function `fillOutScorecard()` is similar to a local variable; it exists while the function is called and ceases to exist when the function exits. If you omit parameters during a function call, the omitted parameters are passed as `undefined`. If you provide extra parameters in a function call that are not required by the function declaration, they are ignored.

Using variables in a function

Local variables are valuable tools for organizing code and making it easy to understand. When a function uses local variables, it can hide its variables from all other scripts in the SWF file; local variables are scoped to the body of the function and cease to exist when the function exits. Any parameters passed to a function are also treated as local variables.

You can also use global and regular variables in a function. However, if you modify global or regular variables, it is good practice to use script comments to document these modifications.

Returning values from a function

Use the `return` statement to return values from functions. The `return` statement stops the function and replaces it with the value of the `return` statement. The following rules govern how to use the `return` statement in functions:

- If you specify a return type other than Void for a function, you must include a `return` statement followed by the returned value in the function.
- If you specify a return type of Void, you generally do not need to include a `return` statement.
- No matter what the return type, you can use a return statement to exit from the middle of a function, provided the return statement is followed by a return value, according to the previous rules.
- If you don't specify a return type, including a `return` statement is optional. If you don't include one, an empty string is returned.

For example, the following function returns the square of the parameter `x` and specifies that the returned value must be a Number:

```
function sqr(x:Number):Number {
  return x * x;
}
```

Some functions perform a series of tasks without returning a value. For example, the following function initializes a series of global variables:

```
function initialize() {
  boat_x = _global.boat._x;
  boat_y = _global.boat._y;
  car_x = _global.car._x;
  car_y = _global.car._y;
}
```

Calling a user-defined function

You can use a target path to call a function in any Timeline from any Timeline, including from the Timeline of a loaded SWF file. If a function was declared using the `_global` identifier, you do not need to use a target path to call it.

To call a function, enter the target path to the name of the function, if necessary, and pass any required parameters inside parentheses. For example, the following statement invokes the function `sqr()` in the movie clip `mathLib` on the main Timeline, passes the parameter 3 to it, and stores the result in the variable `temp`:

```
var temp:Number = this.mathLib.sqr(3);
```

The following example uses an path to call the `initialize()` function that was defined on the main Timeline and requires no parameters:

```
this.initialize();
```

The following example uses a relative path to call the `list()` function that was defined in the `functionsClip` movie clip:

```
this._parent.functionsClip.list(6);
```

CHAPTER 3
Using Best Practices

It is important for Macromedia Flash designers and developers to build applications or animations while being concerned about writing code and structuring applications in a way that is beneficial either to themselves, or to the other people who might work on the project with them. Because Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 are powerful and complex programs, this complexity can sometimes create an uncertainty on how to form a document, format code, or organize ActionScript in a project.

This chapter covers the following topics:

Because it is common for more than one designer or developer to work on a single Flash project, teams benefit when everyone follows a standard set of guidelines for using Flash, organizing FLA files, and writing ActionScript code. The sections in this chapter outline the best practices for using Flash and writing ActionScript. Following guidelines also encourages consistency for people learning how to use Flash and write ActionScript. You should adopt best practices at all times, whether you are a designer or developer, or working alone or as part of a team.

Therefore, it is particularly useful to follow guidelines in the following situations, and for the following reasons:

When working on FLA documents Adopting consistent and efficient practices helps you speed up your workflow. It is faster to develop using best practices, and easier to understand and remember how you structured your document when you edit it further. Additionally, your code is often more portable within the framework of a larger project, and easier to reuse.

When sharing FLA documents Other people editing the document can quickly find and understand ActionScript, consistently modify code, and find and edit assets.

When working on projects Multiple authors can work on a project with fewer conflicts and greater efficiency. Project or site administrators can manage and structure complex projects with fewer conflicts or redundancies if you follow best practices.

When learning or teaching Flash Learning how to build Flash documents using best practices reduces the need to relearn particular methodologies. If students learning Flash practice consistent and better ways to structure code, they might learn the language more quickly and with less frustration. If teachers and authors write using consistent practices and terminology, it is easier for people in the community to learn how to use the program.

Consistent techniques and the following guidelines help those learning Flash or those working in team environments. Consistent methods help you remember how you structured your document when you work alone, particularly if you have not worked on the FLA file recently.

These are only a few of the reasons to learn and follow best practices. There are many other reasons that you are sure to discover when you read these best practices and develop your own good habits. Consider the following sections as a guideline when you are working with Flash; you might choose to follow some or all of the recommendations. You can also modify the recommendations to suit the way you work. The most important thing you can do when working with Flash is to maintain consistency in how you create your documents, which helps you work more efficiently. Many of the guidelines in this chapter help you develop a consistent way of working with Flash and writing ActionScript.

Working with FLA files

Creating FLA files that are consistent makes it easier to work with Flash on a regular basis. It is not uncommon to forget where you have placed particular elements, or what object instance code is placed on. Therefore, follow these guidelines to help you avoid frustrating issues.

Organizing Timelines and the library

Frames and layers on a Timeline are two important parts of the Flash authoring environment. These areas show you where assets are placed and determine how your document works. How a Timeline and the library are set up and used affect the entire FLA file and its overall usability. The following guidelines help you author content efficiently, and let other authors who use your FLA documents have a greater understanding of how the document is structured:

- Give each layer an intuitive layer name, and place related assets together in the same location. Avoid using the default layer names (such as Layer 1, Layer 2), because it can be confusing to remember or locate assets when you are working on complex files.

- Clearly describe the purpose or content of each layer or folder when you name them in a FLA file. This helps users to quickly understand where particular assets are found in layers or folders. It is a good and common practice to name the layer that contains your ActionScript *actions* and to use layer folders to organize similar layers.
- If applicable, place your layers that include ActionScript and a layer for frame labels at the top of the layer stack in the Timeline. This makes it easy to locate the layers that include code and labels.
- Add frame labels in a FLA file instead of using frame numbers in your ActionScript to reference points on the Timeline. This is important and useful if you reference frames in your ActionScript and those frames change later when you edit the Timeline. If you use frame labels and move them on the Timeline, you do not have to change any references in your code.
- Lock your ActionScript layer immediately so that symbol instances or media assets are not placed on that layer. Never put any instances or assets on a layer that includes ActionScript, which can potentially cause conflicts between assets on the Stage and ActionScript that references them.
- Lock layers that you are not using or do not want to modify.
- Use folders in the library to organize similar elements (such as symbols and media assets) in a FLA file. If you name library folders consistently each time you create a file, it is much easier to remember where you put assets. Commonly used folder names are Buttons, MovieClips, Graphics, Assets, Components, and, sometimes, Classes.

Using scenes

Using scenes is similar to using several SWF files together to create a larger presentation. Each scene has a Timeline. When the playhead reaches the final frame of a scene, the playhead progresses to the next scene. When you publish a SWF file, the Timeline of each scene combines into a single Timeline in the SWF file. After the SWF file compiles, it behaves as if you created the FLA file using one scene. Because of this behavior, avoid using scenes for the following reasons:

- Scenes can make documents confusing to edit, particularly in multiauthor environments. Anyone using the FLA document might have to search several scenes within a FLA file to locate code and assets. Consider loading content or using movie clips instead.
- Scenes often result in large SWF files. Using scenes encourages you to place more content in a single FLA file and hence, larger documents to work with and larger SWF files.
- Scenes force users to progressively download the entire SWF file, even if they do not plan or want to watch all of it. Your user progressively downloads the entire file, instead of loading the assets they actually want to see or use. If you avoid scenes, the user can control what content they download as they progress through your SWF file. This means that the user has more control over how much content they download, which is better for bandwidth management. One drawback is the requirement for managing a greater number of FLA documents.
- Scenes combined with ActionScript might produce unexpected results. Because each scene Timeline is compressed onto a single Timeline, you might encounter errors involving your ActionScript and scenes, which typically requires extra, complicated debugging.

There are some situations where few of these disadvantages apply, such as when you create lengthy animations, which is a good time to use scenes. If disadvantages apply to your document, consider using screens to build an animation instead of using scenes. For more information on using screens, see "Creating a new screen-based document (Flash Professional only)" in *Using Flash*.

Saving and version control

When you save your FLA files, it is important to consider using a naming scheme for your documents. Most importantly, use a consistent naming scheme. This is particularly important if you save multiple versions of a single project.

Use intuitive names for your files that are easy to read. Do not use spaces, capitalization, or special characters. Only use letters, numbers, dashes, and underscores. If you save multiple versions of the same file, devise a consistent numbering system such as site_menu01.swf, site_menu02.swf and so on. Many designers and developers choose to use all lowercase characters in their naming schemes. Many Flash designers and developers adopt a naming system that uses a noun-verb or adjective-noun combination for naming files. Two examples of naming schemes are as follows: class_planning.swf and my_project.swf. Avoid cryptic file names.

It is good practice to save new versions of a FLA file when you build an extensive project. The following are different ways that you can save new versions of files:

- Select File > Save As, and save a new version of your document.
- Use version control software (such as SourceSafe, CVS, or Subversion) to control your Flash documents.

Note: SourceSafe on Windows is the only supported version control software that integrates with the Project panel. You can use other version control software packages with FLA documents, but not necessarily in the Project panel. For more information on using Flash Projects and version control, see "Using projects" on page 128.

Some problems might occur if you only work with one FLA file and do not save versions during the process of creating the file. It is possible that files might bloat in size because of the history that's saved in the FLA file, or corrupt (as with any software you use) while you are working on the file. If any of these unfortunate events occur, you will have other versions of your file to use if you save multiple versions throughout your development.

You might also encounter problems when you create an application. Perhaps you made a series of changes to the file, and you do not want to use these changes. Or, you might delete parts of the file that you want to use later in your development. If you save multiple versions while developing, you have an earlier version available if you need to revert.

There are several options that you can use to save a file: Save, Save As, and Save and Compact. When you save a file, Flash does not analyze all the data before creating an optimized version of the document. Instead, the changes you make to the document are appended to the end of the FLA file's data, which shortens the time it takes to save the document. When you select Save As and type a new name for the file, Flash writes a new and optimized version of the file, which results in a smaller file size. When you select Save and Compact, Flash creates a new optimized file and deletes the original file.

Caution: When you select Save and Compact, you cannot undo any changes you made before you saved the file. If you select Save when working with a document, you can undo prior to that save point. Because Save and Compact deletes the earlier version of the file and replaces it with the optimized version, you cannot undo earlier changes.

Remember to frequently use Save As and type a new file name for your document after every milestone in your project if you are not using version control software to create backups of your FLA file. If you encounter major problems while working on the document, you have an earlier version to use instead of losing everything.

There are many software packages that allow users to use version control with their files, which enables teams to work efficiently and reduce errors (such as overwriting files or working on old versions of a document). Popular version control software programs include CVS, Subversion, and SourceSafe. As with other documents, you can use these programs to organize the Flash documents outside Flash.

General coding conventions

Typically, 80% of your development time is spent debugging, troubleshooting, and practicing general maintenance, especially on larger projects. Even when you work on small projects, a significant amount of time is usually spent analyzing and fixing code. When you are working on large-scale projects, you might work with a team. The readability of your code is important for your benefit and the benefit of your team members. When following conventions, you increase readability, which increases workflow and helps find and fix any errors in your code, and all programmers follow a standardized way of writing code, which improves the project in many ways.

The following sections contain several suggestions and guidelines for general coding practices that help improve your workflow and help teams working on a project. This section describes general coding conventions that are true for many programming languages; "ActionScript coding standards" on page 82 discusses conventions that are specific to ActionScript.

Note: Flash Player 7 closely follows the ECMA-262 Edition 4 proposal. It is useful to see this proposal for information on how the language works. (See www.mozilla.org/js/language/es4/index.html.)

General naming guidelines

This section covers naming guidelines for ActionScript and classes. The naming guidelines are detrimental to creating logical code: the primary purpose is to improve readability of your ActionScript. How you name a file can describe the item or process but not refer to its implementation. Avoid using nondescriptive names for methods or variables. For example, if you retrieve a piece of data that is the visitor's username, you might use `getUserName()` instead of the less descriptive `getData()`. This example expresses what is happening rather than how you accomplish it. It is also a good idea to keep all names as short as possible.

Note: All variables must have unique names. Although names are case-sensitive in Flash MX 2004, do not use the same name with a different case because this can be confusing to programmers reading your code and cause problems in earlier versions of Flash that do not force case sensitivity.

Limit your use of abbreviations, but use them consistently. An abbreviation must clearly stand for only one thing. For example, the abbreviation *sec* might represent section and second. Make sure you use it only for one term, and choose a different abbreviation for the other term.

Many developers concatenate words to create names. Use mixed casing when you concatenate words to distinguish between each word for readability. For example, write **myPelican** rather than **mypelican**.

Consider your own naming conventions that are easy to remember, and adapt them for situations where you have common operations you are using. For more information on naming conventions and guidelines, see the following topics:

Variable names

Variable names can only contain letters, numbers, and dollar signs (`$`). Do not begin variable names with numbers. Variables must be unique and they are case-sensitive in Flash Player 7 and earlier. For example, avoid the following variable names:

```
my/warthog = true;   //includes a backslash
my warthogs = false;  //includes a space
5warthogs = 55;    //begins with a number
```

Do not use words that are part of the ActionScript language as variable names. In particular, never use keywords as instance names. For a list of keywords you should not use, see "Avoiding reserved words" on page 75.

Note: Explicitly declare variables with `var` before you use them. Local variables are faster to access.

Avoid using variables that are parts of common programming constructs, even if Flash Player does not include or support the constructs. Because ActionScript is ECMAScript-compliant, application authors can use current ECMA specifications and proposals for information about language structure. For a complete list of reserved or commonly used words, see "Avoiding reserved words" on page 75.

For example, do not use the following keywords as variables:

```
var = "foo";
return = "bar"
interface = 10
TextField = "myTextField";
```

```
switch = true;
new = "funk";
```

Always *strict data type* your variables, which helps avoid errors in your code and trigger code completion. For more information on strict typing and suffixes, see "Using code completion and suffixes" on page 78. You should use suffixes with your variables because suffixes improve ActionScript readability.

Try to keep variables as short as possible while retaining clarity. For more information on processing and performance, see "Performance and Flash Player" on page 114. Some developers use one character variables for temporary variables (such as *i*, *j*, *k*, *m*, and *n*). Use single characters only in certain cases, such as the following:

```
var font_array:Array = TextField.getFontList()
font_array.sort();
for (var i = 0; i<font_array.length; i++) {
  trace(font_array[i]);
}
```

Start variables with a lowercase letter, and use mixed case for concatenated words. If you do not use mixed case, then consistently use lowercase for all your variables. You can also use underscores to separate concatenated words. Do not use a combination of these different naming conventions in a single project.

Some developers use *Hungarian notation* to indicate the data type. Hungarian notation adds a prefix before the variable name to identify what data type the variable is. This form of notation is not recommended as a best practice. However, some developers use it as an alternative to adding suffixes, so you should be aware of the practice. The following ActionScript uses Hungarian notation to indicate the String data type:

```
var strMyName:String = "Rudy";  //indicates string
```

Use complementary pairs when you create a related set of variable names. For example, you might use complementary pairs to indicate a minimum and maximum game score:

```
var minScore:Number = 10; //minimum score
var maxScore:Number = 500; //maximum score
```

The ActionScript editor and Script pane in the Actions panel have built-in support for code completion. This means that Flash shows methods and properties for the current object when you enter ActionScript code. You must name variables in a particular way or use specific commenting techniques to see code completion menus. The following are the three ways to use code completion:

- Use strict data typing, which is recommended if you are using ActionScript 2.0.
- Use suffixes, which is recommended if you are using ActionScript 1.
- Use comments, which is recommended only if you do not use suffixes or strict data typing.

For more information on code completion, see "Using code completion and suffixes" on page 78.

Constants

Variables should be lowercase or mixed-case letters; however, constants (variables that do not change) should be uppercase. Separate words with underscores, as the following ActionScript shows:

```
var BASE_URL:String = "http://www.macromedia.com";  //constant
var MAX_WIDTH:Number = 10;  //constant
```

Write static constants in uppercase, and separate words with an underscore. Do not directly code numerical constants unless the constant is `1`, `0`, or `-1`, which you might use in a `for` loop as a counter value.

You can use constants for situations in which you need to refer to a property whose value never changes. This helps you find typographical mistakes in your code that might not be found if you used literals. It also lets you change the value in a single place.

Functions

Function names start with a lowercase letter. Describe what value is being returned when you create function names. For example, if you are returning the name of a song title, you might name the function `getCurrentSong()`.

Establish a standard for relating similar functions, because ActionScript does not permit overloading. In the context of Object Oriented Programming (OOP), *overloading* refers to the ability to make your functions behave differently depending on what data types are passed into it.

Methods

Name methods as verbs with mixed case for concatenated words, making sure that the first letter is lowercase. For example, you might name methods in the following ways:

```
sing();
boogie();
singLoud();
danceFast();
```

You use verbs for most methods because they perform an operation on an object.

Loops

Loop indexes (such as a `for` loop) have standardized naming conventions. A simple index loop (such as a `for` loop or `while` loop) typically uses `i`, `j`, `k`, `m`, and `n` as control variables. Use these single-character variable names only for short loop indexes, or when performance optimization and speed are critical.

Classes and objects

Class names are usually nouns or qualified nouns, beginning with an uppercase first letter. A qualifier describes the noun or phrase. For example, instead of member, you might qualify the noun using `NewMember` or `OldMember`. Sometimes a class name is a compound word. Write class names in mixed case beginning with an uppercase letter when the name includes concatenated words. You do not have to pluralize the words you use in the class name, and, in most cases, it is better to leave the words as qualified nouns.

Try to make the class name descriptive and simple, but most importantly, select a meaningful class name. The name is descriptive of the class's contents: do not be vague or misleading when you name a class. Try to avoid acronyms and abbreviations unless they are more commonly used than the long form (such as HTML), and remember that clear names are more important than short names.

Do not use a class name in the properties of that class because it causes redundancy. For example, it does not make sense to have `Cat.catWhiskers`. Instead, `Cat.whiskers` is much better. Do not use nouns that also might be interpreted as verbs, which might lead to confusion with methods, states, or other application activities. Select meaningful, specific names rather than generic names.

See the following examples of class names for proper formatting:

```
class Widget;
class PlasticWidget;
class StreamingVideo;
```

It is a good practice to try and communicate the relationship a class has within a hierarchy when you name it. This helps display its relationship within an application. For example, you might have the `Widget` interface, and the implementation of `Widget` might be `PlasticWidget`, `SteelWidget`, and `SmallWidget`. For information on interfaces, see "Interfaces" on page 249. For information on implementations, see `class` in *Flash ActionScript Language Reference*.

You might have public and private member variables in a class. The class can contain variables that you do not want users to set or access directly. Make these variables private and only allow users to access the values using getter/setter methods. When naming member variables in your class files, it is advisable and common practice to prefix the variable names with `m_`. This helps flag the variables as belonging to the class you're creating, and generally makes the code more readable.

Set most member variables to private unless there is a good reason for making them public. It is much better from a design standpoint to make member variables private and allow access only to those variables through a group of getter/setter functions. For more information on using private and public member variables, see "Controlling member access" on page 256.

Packages

Put the prefix for a package name in all lowercase letters. Begin package names with `mx` or `com.macromedia` to maintain consistency when naming classes. The next parts of the package name vary, depending on the particular naming scheme. For example, a convention might use one of the following package names:

```
mx.containers.ScrollPane
```

```
// or
mx.containers.TextInput
```

This convention uses a prefix of `mx.containers`.

If you create your own packages, use a consistent naming convention, such as starting with `com.macromedia.PackageName`. Or, use a convention of naming your packages `com.yourcompany.PackageName`.

A clear and self-explanatory package name is important because it explains the package's responsibilities. For example, you might have a package named `Shapes`, which is responsible for drawing various kinds of geometric shapes using the Flash drawing API; its name would be `com.macromedia.Shapes`.

Interfaces

Interface names have an uppercase first letter, the same as class names. Interface names are usually adjectives, such as `Printable`. The following interface name, `EmployeeRecords`, uses an initial uppercase letter and concatenated words with mixed case:

```
interface EmployeeRecords{}
```

Note: Some developers start interface names with an uppercase `I` to distinguish them from classes.

Components and linkage

Component names have an uppercase first letter, and any concatenated words are written in mixed case. For example, the following default UI component set uses concatenated words and mixed case:

- CheckBox
- ComboBox
- DataGrid
- DateChooser
- DateField
- MenuBar
- NumericStepper
- ProgressBar
- RadioButton
- ScrollPane
- TextArea
- TextInput

Components not using concatenated words begin with an uppercase letter.

If you develop custom components, use a naming convention to prevent naming collisions with Macromedia components. The names of your components must be different from those of the default set that is included with Flash. Adopting your own consistent naming convention prevents naming conflicts.

For more information on Component conventions, see"Working with components in Flash Player" on page 115.

Remember that the naming conventions in this section are guidelines. It is most important to use a naming scheme that works well for you and use it consistently.

Avoiding reserved words

Avoid using reserved words as instance names in Flash. As a general rule, avoid using any word in ActionScript as an instance or variable name. The following table lists reserved keywords in Flash that cause errors in your scripts:

add	and	break	case
catch	class	continue	default
delete	do	dynamic	else
eq	extends	false	finally
for	function	ge	get
gt	if	ifFrameLoaded	implements
import	in	instanceof	interface
intrinsic	le	it	ne
new	not	null	on
onClipEvent	or	private	public
return	set	static	super
switch	tellTarget	this	throw
try	typeof	var	void
while	with		

The following words are reserved for future use in Flash, from the ECMA-262 specification, and ECMA-262 Edition 4 proposal. Avoid using these words because they might be used in future editions of Flash.

as	abstract	Boolean	bytes
char	const	debugger	double
enum	export	final	float
goto	is	long	namespace
native	package	protected	short
synchronized	throws	transient	use
volatile			

All class names, component class names, and interface names are also reserved words, as listed in the following table:

Accessibility	Accordion	Alert	Array
Binding	Boolean	Button	Camera
CellRenderer	CheckBox	Collection	Color
ComboBox	ComponentMixins	ContextMenu	ContextMenuItem
CustomActions	CustomFormatter	CustomValidator	DataGrid
DataHolder	DataProvider	DataSet	DataType
Date	DateChooser	DateField	Delta
DeltaItem	DeltaPacket	DepthManager	EndPoint
Error	FocusManager	Form	Function
Iterator	Key	Label	List
Loader	LoadVars	LocalConnection	Log
Math	Media	Menu	MenuBar
Microphone	Mouse	MovieClip	MovieClipLoader
NetConnection	NetStream	Number	NumericStepper
Object	PendingCall	PopUpManager	PrintJob
ProgressBar	RadioButton	RDBMSResolver	Screen
ScrollPane	Selection	SharedObject	Slide
SOAPCall	Sound	Stage	String
StyleManager	System	TextArea	TextField
TextFormat	TextInput	TextSnapshot	TransferObject
Tree	TreeDataProvider	TypedValue	UIComponent
UIEventDispatcher	UIObject	Video	WebService
WebServiceConnector	Window	XML	XMLConnector
XUpdateResolver			

Formatting code

Formatting ActionScript is essential to writing maintainable code. For example, it would be extremely difficult to follow the logic of a FLA file that has no indenting or comments, or has inconsistent formatting and naming conventions. By indenting blocks of code (such as loops and `if` statements), the code is easy to read and debug. For more information on formatting and white space, see "Spacing and readability" on page 81 and "Wrapping lines of code" on page 103. For information on structuring ActionScript syntax, see "Writing syntax and statements" on page 89.

For more information on formatting code, see the following topics:

- "Using comments in code" on page 77
- "Using code completion and suffixes" on page 78
- "Using recommended suffixes" on page 80
- "Spacing and readability" on page 81

Using comments in code

One of most important aspects of any project, whether it is a simple widget or a large scale application, is documentation. Without comments, it is likely you will not know why code was written or organized in a certain way. For this reason, it is important to thoroughly document your code in key points of the application. For example, if you must write a certain workaround for a complicated situation, document what you are doing and why. Whoever works with the code in the future can understand what is happening and might not inadvertently break that code.

Consistently use comments in your ActionScript, and describe what the code is doing. Using comments is useful to help you remember coding decisions, and it is extremely helpful for anyone else reading your code. Comments must clearly explain the intent of the code and not just translate the code. If something is not readily obvious in the code, you should add comments to it.

Avoid using *cluttered comments*, despite how popular they sometimes are to add. An example of cluttered comments is a line of equal signs (=) or asterisks (*) used to create a block or separation around your comments. Instead, use white space to separate your comments from the ActionScript. If you format your ActionScript using the Auto Format tool, this removes the white space. Remember to add it back in or use single comment lines (//) to maintain spacing; these lines are easy to remove after you format your code.

Before you deploy your project, remove any superfluous comments from the code. If you find that you have many comments in the ActionScript, consider whether you need to rewrite some of the ActionScript. If you feel you must include many comments about how the code works, it is usually a sign of inelegant ActionScript.

Note: Using comments is most important in ActionScript that is intended to teach an audience. For example, add comments to your code if you are creating sample applications for the purpose of teaching Flash, or if you are writing documentation on ActionScript.

Comments document the decisions you make in the code, answering both *how* and *why*. For example, you might describe a workaround in comments. Therefore, the related code is easily found for updating or fixing at a later date, by another developer, or if the issue is addressed in a future version of Flash or Flash Player and the workaround is no longer necessary.

Use block comments for multiline comments, as the following example shows:

```
/*
  The following ActionScript initializes variables used in the main and sub-
  menu systems. Variables are used to track what options are clicked.
*/
```

Note: If you place the comment characters (`/*` and */) on separate lines at the beginning and end of the comment, you can easily comment them out by placing double slash characters (//) in front of them (for example, ///* and //*/). This lets you quickly and easily comment and uncomment your code.

Use the following format for single-line comments:

```
// the following sets a local variable for age
var myAge:Number = 26;
```

The following single-line comment is formatted as a trailing comment, which is on the same line as the ActionScript:

```
var myAge:Number = 26; //variable for my age
```

If you use the Auto Format feature with your code, trailing comments move to the next line. Add these comments after you format your code, or you must modify their placement after using Auto Format.

Sometimes you must distinguish important comments when writing code. The following conventions demonstrate some of the common methodologies that programmers use.

When more code needs to be added to a section, you can use the following format:

```
// :TODO: comment
```

Use the following format when there is a known issue with your code or application, or there is another problem. Note a bug's ID or number, if possible. Adding a bug ID helps prevent recurring problems and saves time.

```
// :BUG: [bug id] comment
```

When ActionScript needs further modifications because it is inelegant or does not follow best practices, use the following format:

```
// :KLUDGE:
```

When ActionScript is intricately designed and interacts with other areas of an application, use the following format to alert developers who might work on the code later:

```
// :TRICKY:
```

These practices help Flash users who learn from the commented source code. Another benefit of using these methodologies is that you can easily locate bugs and unfinished items by using the Find tool in the Actions panel.

Note: For readability, single-line comments (//) are often used instead of block comments. This increases the readability of the code for instructional purposes.

For information on using comments in classes, see "Using comments in classes" on page 102.

Using code completion and suffixes

Use strict data typing with your variables whenever possible because it helps you in the following ways:

- Adds code completion functionality, which speeds up coding.
- Helps you avoid errors in your compiled SWF files.

To strict data type your variables, you must define the variable using the `var` keyword. In the following example, when creating a LoadVars object, you would use strict data typing:

```
var params_lv:LoadVars = new LoadVars();
```

Strict data typing provides you with code completion, and ensures that the value of `params_lv` contains a LoadVars object. It also ensures that the LoadVars object is not used to store numeric or string data. Because strict typing relies on the `var` keyword, you cannot add strict data typing to global variables or parameters within an Object or array. For more information on strict typing variables, see "Strict data typing" on page 41.

Note: Strict data typing does not slow down a SWF file. Type checking occurs at compile time (when the SWF file is created), not at runtime.

Adding *suffixes* to your variable names serves the following important functions:

- It provides valuable code completion, which helps speed up the coding process.
- It makes your code readable, so you can immediately identify a variable's data type.

For example, the following code creates an array that contains the names of each font installed on the client computer. This code demonstrates the use of code suffixes (`_array`) and strict typing (`:Array`). Suffixes and strict typing let you use code completion, but strict typing ensures that a variable maintains its data type. Suffixes improve the readability of your ActionScript; for example, it is easier to understand what `font_array` means, but `font` might not be as obvious in the code. In the following example, if the code snippet tries to assign a string or numeric value to `font_array`, an error is generated:

```
var font_array:Array = TextField.getFontList();
```

When you write instructional code, use suffixes to improve the code's readability for students. Suffixes help users learn ActionScript because they clarify each variable's data type, which helps explain the code's structure.

You can also generate code completion by using a specific comment technique. If you want to add code completion for an object within your ActionScript, add the following comment to your code:

```
// Object siteParams_obj;
```

Whenever you enter **siteParams_obj.** (with the dot [.]) into the Actions panel, the code completion menu for your Object appears. Using comments to provide code completion is the most obscure method; use it only as a last resort for code completion, such as when you need to create backward-compatible files. The best and recommended methods for code completion and readability are strict typing and suffixes, respectively.

To use code completion with components, you must import the component's class, or provide the full path to the ActionScript class name. Format your ActionScript in one of the following ways:

- You can specify its fully qualified class name, as the following example shows:

  ```
  var myScrollPane:mx.containers.ScrollPane;  //code completion works
  ```

- You can also use the `import` statement to reference the class, as the following example shows:

  ```
  import mx.containers.ScrollPane;
  var myScrollPane:ScrollPane;  //code completion works
  ```

However, the following ActionScript throws an error and does not allow you to use code completion:

```
var myScrollPane:ScrollPane;  //no code completion, throws error
```

Using recommended suffixes

You can use the following suffixes to generate code completion in the Actions panel. They are recommended because suffixes encourage consistency in code that might be shared among Flash authors. Although suffixes are not necessary for code completion in ActionScript 2.0 documents, using consistent suffixes helps other developers understand how your ActionScript works.

Object	Suffix
Array	_array
Button	_btn
Camera	_cam
Color	_color
ContextMenu	_cm
ContextMenuItem	_cmi
Date	_date
Error	_err
LoadVars	_lv
LocalConnection	_lc
Microphone	_mic
MovieClip	_mc
MovieClipLoader	_mcl
NetConnection	_nc
NetStream	_ns
PrintJob	_pj
SharedObject	_so
Sound	_sound
String	_str
TextField	_txt
TextFormat	_fmt
Video	_video

Use the following suffixes with component instances. These suffixes *do not* generate code completion in the Actions panel. However, using suffixes encourages consistency in ActionScript that might be shared among developers.

Object	Suffix
Accordion	_acc
Alert	_alert
Button	_button
CheckBox	_ch
ComboBox	_cb
DataGrid	_dg
DateChooser	_dc
DateField	_df
Label	_label
List	_list
Loader	_ldr
Menu	_menu
MenuBar	_mb
NumericStepper	_nstep
ProgressBar	_pb
RadioButton	_rb
ScrollPane	_sp
TextArea	_ta
TextInput	_ti
Tree	_tr
Window	_win

Spacing and readability

Use spaces, line breaks, and tab indents to increase the readability of your code. Readability increases by showing the code hierarchy, which you can emphasize with spacing. Spacing and code readability are important because they make ActionScript easier to understand. This is important for students learning ActionScript as well as experienced users working on complex projects. Legibility is important when you are debugging ActionScript, because it is much easier to spot errors when code is formatted correctly and properly spaced.

Put one blank line in between paragraphs of ActionScript Paragraphs of ActionScript are groups of logically related code. By breaking the code into modules, it helps users reading the ActionScript understand its logic. For more information on spacing in classes and statements, see "Wrapping lines of code" on page 103.

Use line breaks to make complex statements easier to read You can format some statements, such as conditional statements, in several ways. Sometimes, formatting statements across several lines rather than a single line makes it easier to read. For more information on properly formatting statements, see "Writing syntax and statements" on page 89.

Use consistent indentation in your code Indenting helps show the hierarchy of the code's structure. Use the same indentation throughout your ActionScript, and make sure that you align the braces ({ }) properly. Aligned braces improve the readability of your code. If your ActionScript syntax is correct, Flash automatically indents the code correctly when you press Enter (Windows) or Return (Macintosh). You can also press the Auto Format button in the Actions panel to indent your ActionScript if the syntax is correct.

Note: You can control autoindentation and indentation settings by selecting Edit > Preferences, and then selecting the ActionScript tab.

ActionScript coding standards

One of the most important aspects about programming is consistency, whether it relates to variable naming schemes, coding standards, or where you place your ActionScript code. Code debugging and maintenance is dramatically simplified if the code is organized and adheres to standards. Formatted code that follows an established set of guidelines is easier to maintain, and easier for other developers to understand and modify.

For more information, see the following topics:

- "Organizing ActionScript in a document" on page 82
- "Writing ActionScript" on page 85
- "Using scope" on page 95
- "Using functions" on page 98

Organizing ActionScript in a document

It is important to understand where to put your ActionScript: Should it be in the FLA file, should it be put on the server in an external AS file, or should it be a class written using ActionScript 2.0? The first thing you must understand is how to structure your project. If you are building an application, read "Building Flash Applications" on page 121.

A general guideline is to put all your code in as few places as possible, whether you put it inside a FLA document or in external files. For information on choosing between ActionScript 1 or ActionScript 2.0, see "Choosing between ActionScript 1 and ActionScript 2.0" on page 84. Organizing your code helps you edit projects more efficiently, because you can avoid searching in different places when you debug or modify the ActionScript. The following sections provide more information about where to put your ActionScript.

Keeping actions together

Whenever possible, put your ActionScript in a single location. If you put code in a FLA file, put ActionScript on the first or second frame on the Timeline, in a layer called *actions* that is the first or second layer on the Timeline. Sometimes you might create two layers for ActionScript to separate functions, which is an acceptable practice. Some Flash applications do not always put all your code in a single place (in particular, when you use screens or behaviors). For more information on organizing ActionScript in an application, see "Organizing files and storing code" on page 125. For more information on design patterns, see "Using design patterns" on page 104 and "Using the MVC design pattern" on page 126.

Despite these rare exceptions, you can usually put all your code in the same location. The following are the advantages of this process:

- Code is easy to find in a potentially complex source file.
- Code is easy to debug.

One of the most difficult parts of debugging a FLA file is finding all the code. After you find all the code, you must figure out how it interacts with other pieces of code as well as the FLA file. If you put all your code in a single frame, it is much easier to debug because it is centralized, and these problems reduce in number. For information on attaching code to objects (and decentralizing your code), see "Attaching code to objects" on page 83. For information on behaviors and decentralized code, see "Using behaviors" on page 106.

Attaching code to objects

Avoid attaching ActionScript to objects in a FLA file, even in simple SWF files. Attaching code to an object means that you select a movie clip, component, or button instance; open the Actions panel; and add ActionScript using the `on()` or `onClipEvent()` handler functions.

This practice is strongly discouraged for the following reasons:

- ActionScript that is attached to objects is difficult to locate, and the FLA files are difficult to edit.
- ActionScript that is attached to objects is difficult to debug.
- ActionScript that is written on the Timeline or in classes is more elegant and easier to build upon.
- ActionScript that is attached to objects encourages poor coding style.
- ActionScript that is attached to objects forces students and readers to learn different coding styles, additional syntax, and a poor and limited coding style. This can be a frustrating experience.

Some Flash users might say it is easier to learn ActionScript by attaching code to an object because it might be easier to add simple code, or write about or teach ActionScript this way. A significant problem exists because most people learning ActionScript need to know how to write the equivalent code, which is not difficult. The contrast between two styles of coding can also be confusing to people learning ActionScript, which is why consistency throughout the learning process has advantages.

Attaching ActionScript to a button called `myButton_btn` looks like the following ActionScript, and should be avoided:

```
on (release) {
  //do something
}
```

However, placing ActionScript with the same purpose on the Timeline looks like the following code, which is encouraged:

```
myButton_btn.onRelease = function() {
  //do something
};
```

For more information on ActionScript syntax, see "Writing syntax and statements" on page 89.

Note: Different practices apply when using behaviors and screens, which sometimes involves attaching code to objects. For more information and guidelines, see "Comparing timeline code with object code" on page 105 and "Organizing code for screens" on page 108.

Choosing between ActionScript 1 and ActionScript 2.0

When you start a new document or application in Flash, you must decide how to organize its associated files. You might use classes in some projects, such as when you are building applications or complex FLA files, but not all documents use classes. For example, many short examples in the documentation do not use classes. Using classes to store functionality is not the easiest or best solution for small applications or simple FLA files. In these cases, it is often more efficient to put ActionScript inside the document. In this case, try to put all your code on the Timeline on as few frames as possible, and avoid placing code on or in instances (such as buttons or movie clips) in a FLA file.

When you build a small project, it is often more work and effort to use classes or external code files to organize ActionScript instead of adding ActionScript within the FLA file. Sometimes it is easier to keep all the ActionScript within the FLA file, rather than placing it within a class that you import. This does not mean that you should necessarily use ActionScript 1. You might decide to put your code inside the FLA document using ActionScript 2.0 with its strict data typing and its new methods and properties. ActionScript 2.0 also offers a syntax that closely follows standards in other programming languages. This makes the language easier and more valuable to learn. For example, you will feel familiar with ActionScript if you have encountered another language that's based on the same structure and syntax standards. Or, you can apply this knowledge to other languages you learn in the future. ActionScript 2.0 lets you use an object-oriented approach to developing applications using an additional set of language elements, which can be advantageous to your application development. For more information on the difference between ActionScript 1 and ActionScript 2.0, see "New object-oriented programming model" on page 21 and "Porting existing scripts to Flash Player 7" on page 13.

There are cases in which you cannot choose which version of ActionScript to use. If you are building a SWF file that targets an old version of Flash Player, such as a mobile device application, you must use ActionScript 1, which is compatible.

Remember, regardless of the version of ActionScript, you should follow good practices. Many of these practices, such as remaining consistent with case sensitivity, using code completion, enhancing readability, avoiding keywords for instance names, and keeping a consistent naming convention, apply to both versions.

If you plan to update your application in future versions of Flash, or make it larger and more complex, you should use ActionScript 2.0 and classes, to make it easier to update and modify your application.

ActionScript and Flash Player

If you compile a SWF file that contains ActionScript 2.0 with Publish Settings set to Flash Player 6 and ActionScript 1, code functions as long as it does not use ActionScript 2.0 classes. There is no case sensitivity or strict data typing involved with the code. However, if you compile your SWF file with Publish Settings set to Flash Player 7 and ActionScript 1, strict data typing and case sensitivity are enforced.

ActionScript 2.0 is an improvement to the compiler in the Authoring environment, so you can target earlier versions of Flash Player while working with ActionScript 2.0.

Writing ActionScript

ActionScript is a case-sensitive language. This means that variables can have the same name, because slightly different capitalization is considered two separate instances, as the following ActionScript shows:

```
var firstName:String = "Jimmy";
trace(firstname); //displays: undefined
```

In earlier versions of Flash (Flash MX and earlier), Flash would trace the string `Jimmy` in the Output panel. Because Flash is case-sensitive, `firstName` and `firstname` are two separate variables. This is an important concept to understand. If your earlier FLA files have slightly varying variable capitalization, you might experience unexpected results or broken functionality when converting the file or application for Flash Player 7.

Note: You should not use two variables that differ only in case. If you are using them as separate variables, change the instance name, not just the case.

Case sensitivity can have a large impact when working with a web service that uses its own rules for variable naming and what case variables are in when they are returned to the SWF file from the server. For example, if you use a Macromedia ColdFusion web service, property names from a structure or object might be all uppercase, such as `FIRSTNAME`. Unless you use the exact same case in Flash, you are likely to experience unexpected results.

Try to keep ActionScript in a FLA file as generic as possible for the user interface. It is not ideal to write ActionScript that depends on specific target paths. If the code needs to interact with interface elements in a larger application, try using a model-view-controller based approach. For more information, see “Using the MVC design pattern” on page 126.

For more information on guidelines for writing ActionScript, see the following topics:

Adding initialization

One of the easiest ways to initialize code using ActionScript 2.0 is to use classes. You can encapsulate all your initialization for an instance within the class's constructor function, or abstract it into a separate method, which you would explicitly call after the variable has been created, as the following code shows:

```
class Product {
  function Product() {
    var prod_xml:XML = new XML();
    prod_xml.ignoreWhite = true;
    prod_xml.onLoad = function(success:Boolean) {
      if (success) {
        trace("loaded");
      } else {
        trace("error loading XML");
      }
    };
    prod_xml.load("products.xml");
  }
}
```

The following code could be the first function call in the application, and the only one you make for initialization. Frame 1 of a FLA document that is loading XML might use code that is similar to the following ActionScript:

```
if (init == undefined) {
  var prod_xml:XML = new XML();
  prod_xml.ignoreWhite = true;
  prod_xml.onLoad = function(success:Boolean) {
    if (success) {
      trace("loaded");
    } else {
      trace("error loading XML");
    }
  };
  prod_xml.load("products.xml");
  init = true;
}
```

Using trace statements

Use `trace` statements in your documents to help you debug your code while authoring the FLA file. For example, by using a `trace` statement and `for` loop, you can see the values of variables in the Output panel, such as strings, arrays, and objects, as the following example shows:

```
var day_array:Array = ["sun", "mon", "tue", "wed", "thu", "fri", "sat"];
var numOfDays:Number = day_array.length;
for (var i = 0; i<numOfDays; i++) {
  trace(i+": "+day_array[i]);
}
```

This displays the following information in the Output panel:

```
0: sun
1: mon
2: tue
3: wed
4: thu
5: fri
6: sat
```

Using a `trace` statement is an efficient way to debug your ActionScript.

You can remove your `trace` statements when you publish a SWF file, which makes minor improvements to playback performance. Before publishing a SWF file, open Publish Settings and select Omit Trace Actions on the Flash tab. For more information on using trace, see `trace()` in *Flash ActionScript Language Reference*.

Using the super prefix

If you refer to a method in the parent class, prefix the method with `super` so that other developers know from where the method is invoked. The following ActionScript demonstrates the use of proper scoping using the `super` prefix:

```
// readable
var pelican = super.bird;
var animal = super.super.bird;
```

The following ActionScript is not as readable. Avoid using the following syntax because it does not reveal from where you invoke the method:

```
// not readable
var pelican = bird; // scoped to super.bird
var animal = super.super.bird;
```

Avoiding the with statement

One of the more confusing concepts for people learning ActionScript to understand is using the `with` statement. Consider the following code that uses the `with` statement:

```
this.attachMovie("circle_mc", "circle1_mc", 1);
with (circle1_mc) {
  _x = 20;
  _y = Math.round(Math.random()*20);
  _alpha = 15;
```

```
  createTextField("label_txt", 100, 0, 20, 100, 22);
  label_txt.text = "Circle 1";
  someVariable = true;
}
```

In this code, you attach a movie clip instance from the library and modify its properties using the `with` statement. When you do not specify a variable's scope, you do not always know where you are setting properties, so your code can be confusing. In the previous code, you might expect `someVariable` to be set within the `circle1_mc` movie clip, but it is actually set in the main Timeline of the SWF file.

It is easier to follow what is happening in your code if you explicitly specify the variables scope, instead of relying on the `with` statement. The following example shows a slightly longer, but better, ActionScript example that specifies the variables scope:

```
this.attachMovie("circle_mc", "circle1_mc", 1);
circle1_mc._x = 20;
circle1_mc._y = Math.round(Math.random()*20);
circle1_mc._alpha = 15;
circle1_mc.createTextField("label_txt", 100, 0, 20, 100, 22);
circle1_mc.label_txt.text = "Circle 1";
circle1_mc.someVariable = true;
```

There is an exception to this rule. When you are working with the drawing API to draw shapes, you might have several similar calls to the same methods (such as `lineTo` or `curveTo`) because of the drawing API's functionality. For example, when drawing a simple rectangle, you need four separate calls to the `lineTo` method, as the following code shows:

```
this.createEmptyMovieClip("rectangle_mc", 1);
with (rectangle_mc) {
  lineStyle(2, 0x000000, 100);
  beginFill(0xFF0000, 100);
  moveTo(0, 0);
  lineTo(300, 0);
  lineTo(300, 200);
  lineTo(0, 200);
  lineTo(0, 0);
  endFill();
}
```

If you wrote each `lineTo` or `curveTo` method with a fully qualified instance name, the code would quickly become cluttered and difficult to read and debug.

Using variables

For the initial values for variables, assign a default value or allow the value of `undefined`, as the following class example shows. This class sets the initial values of `m_username` and `m_password` to empty strings:

```
class User {
  private var m_username:String = "";
  private var m_password:String = "";
  function User(username:String, password:String) {
    this.m_username = username;
    this.m_password = password;
```

```
    }
}
```

Delete variables or make variables `null` when you no longer need them. Setting variables to `null` can still enhance performance. This process is commonly called *garbage collection*. Deleting variables helps optimize memory use during runtime, because unneeded assets are removed from the SWF file. It is better to delete variables than to set them to `null`. For more information on performance, see "Performance and Flash Player" on page 114.

For information on naming variables, see "Variable names" on page 70. For more information on deleting objects, see `delete` in *Flash ActionScript Language Reference*.

Writing syntax and statements

There are several ways to format or write a piece of ActionScript. Differences can exist in the way you form the syntax, such as the way you form it across multiple lines in the Actions panel (for example, where you put brackets ({}) or parentheses [()]).

There are several general guidelines for writing ActionScript syntax. Place one statement per line to increase the readability of your ActionScript. The following example shows correct and incorrect statement usage:

```
theNum++;       //correct
theOtherNum++;  //correct
aNum++; anOtherNum++;  //incorrect
```

There are several ways you can assign variables. Do not embed assignments, which is sometimes used to improve performance in a SWF file at runtime. Despite the performance boost, your code is much harder to read and debug. Consider the following ActionScript example:

```
var myNum = (a = b + c) + d;
```

If you assign variables as separate statements, it improves readability, as the following example shows:

```
var a = b + c;
var myNum = a + d;
```

The following sections describe the preferred ways to format your syntax and statements in ActionScript:

- "Writing conditional statements" on page 90
- "Writing compound statements" on page 91
- "Using the for statement" on page 92
- "Using while and do-while statements" on page 92
- "Using return statements" on page 93
- "Writing switch statements" on page 93
- "Using try-catch and try-catch-finally statements" on page 93
- "Using listener syntax" on page 94

Writing conditional statements

Place conditions put on separate lines in `if`, `else-if`, and `if-else` statements. Your `if` statements should use braces (`{}`). You should format braces like the following examples. The `if`, `if-else`, and `else-if` statements have the following formats:

```
//if statement
if (condition) {
  //statements;
}

//if-else statement
if (condition) {
  //statements;
} else {
  //statements;
}

//else-if statement
if (condition) {
  //statements;
} else if (condition) {
  //statements;
} else {
  //statements;
}
```

You can write a conditional statement that returns a Boolean value, as the following example shows:

```
if (cart_array.length>0) {
  return true;
} else {
  return false;
}
```

However, compared with the previous code, the ActionScript in the following example is preferable:

```
return (cart_array.length > 0);
```

The second snippet is shorter and has fewer expressions to evaluate. It's easier to read and understand.

The following condition checks if the variable *y* is greater than zero, and returns the result of `x/y` or a value of 0:

```
if (y>0) {
  return x/y;
} else {
  return 0;
}
```

The following example shows another way to write this code:

```
return ((y > 0) ? x/y : 0);
```

The shortened `if` statement syntax is known as the conditional operator (`?:`). It lets you convert simple `if-else` statements into a single line of code. In this case, the shortened syntax reduces readability, and so it is not preferable. Do not use this syntax for complex code, because it is more difficult to spot errors. For more information on using conditional operators, see "Writing conditional statements" on page 90 and "Writing compound statements" on page 91.

When you write complex conditions, it is good form to use parentheses [()] to group conditions. If you do not use parentheses, you (or others working with your ActionScript) might run into operator precedence errors.

For example, the following code does not use parentheses around the condition:

```
if (fruit == apple && veggie == leek) {}
```

The following code uses good form by adding parentheses around conditions:

```
if ((fruit == apple) && (veggie == leek)) {}
```

If you prefer using conditional operators, place the leading condition (before the question mark [?]) inside parentheses. This helps improve the readability of your ActionScript. The following code is an example of ActionScript with improved readability:

```
(y >= 5) ? y : -y;
```

Writing compound statements

Compound statements contain a list of statements within braces ({}). The statements within these braces are indented from the compound statement, as the following ActionScript shows:

```
if (a == b) {
  //this code is indented
  trace("a == b");
}
```

In this example, the opening brace is placed at the end of the compound statement. The closing brace begins a line, and aligns with the beginning of the compound statement.

Place braces around each statement when it is part of a control structure (`if-else` or `for`), even if it contains only a single statement. This good practice helps you avoid errors in your ActionScript when you forget to add braces to your code. The following example shows code that is written using poor form:

```
if (numUsers == 0)
  trace("no users found.");
```

Although this code validates, it is considered poor form because it lacks braces around the statements. In this case, if you add another statement after the `trace` statement, it is executed regardless of whether the `numUsers` variable equals 0, which can lead to unexpected results. For this reason, add braces so the code looks like the following example:

```
if (numUsers == 0) {
  trace("no users found");
}
```

Using the for statement

Write the `for` statement using the following format:

```
for (init; condition; update) {
  // statements;
}
```

The following structure demonstrates the `for` statement:

```
for (var i = 0; i<4; i++) {
  myClip_mc.duplicateMovieClip("newClip"+i+"_mc", i+10, {_x:i*100, _y:0});
}
```

Remember to include a space following each expression in a `for` statement.

Using while and do-while statements

Write `while` statements, or empty `while` statements, using one of the following formats:

```
while (condition) {
  //statements;
}
```

or:

```
while (condition);
```

The following ActionScript is an example of a `while` statement:

```
var users_ds:mx.data.components.DataSet;
//
users_ds.addItem({name:'Irving', age:34});
users_ds.addItem({name:'Christopher', age:48});
users_ds.addItem({name:'Walter', age:23});
//
users_ds.first();
while (users_ds.hasNext()) {
  trace("name:"+users_ds.currentItem['name']+",
  age:"+users_ds.currentItem['age']);
  users_ds.next();
}
```

Write `do-while` statements using the following format:

```
do {
  //something
} while (condition);
```

The following is an example of a `do-while` statement:

```
var i:Number = 15;
do {
  trace(i);
  i++;
} while (i<5);
```

Using return statements

Do not use parentheses [()]with any `return` statements that have values. The only time you should use parentheses with `return` statements is when it makes the value more obvious, which is shown in the third line of the following ActionScript:

```
return;
return myCar.paintColor();
// parentheses used to make the return value obvious
return ((paintColor)? paintColor: defaultColor);
```

Writing switch statements

All `switch` statements include a `default` case. The `default` case includes a `break` statement to prevent a fall-through error if another case is added. For example, if the condition in the following example evaluates to `A`, both the statements for case `A` and `B` execute, because case `A` lacks a `break` statement. The `default` case should also be the last case in the `switch` statement. When a case falls through, it does not have a `break` statement, but includes a comment in the `break` statement's place, which you can see in the following example after `case A`. Write `switch` statements using the following format:

```
switch (condition) {
case A :
  //statements
  //falls through
case B :
  //statements
  break;
case Z :
  //statements
  break;
default :
  //statements
  break;
}
```

Using try-catch and try-catch-finally statements

Write `try-catch` and `try-catch-finally` statements using the following formats:

```
//try-catch
try {
  //statements
} catch (myError) {
  //statements
}

//try-catch-finally
try {
  //statements
} catch (myError) {
  //statements
} finally {
  //statements
}
```

Using listener syntax

There are several ways to write listeners for events in Flash MX 2004. Some popular techniques are shown in the following code examples. The first example shows a properly formatted listener syntax, which uses a Loader component to load content into a SWF file. The `progress` event starts when content loads, and the `complete` event indicates when loading finishes.

```
var box_ldr:mx.controls.Loader;
var ldrListener:Object = new Object();
ldrListener.progress = function(evt:Object) {
  trace("loader loading:"+Math.round(evt.target.percentLoaded)+"%");
};
ldrListener.complete = function(evt:Object) {
  trace("loader complete:"+evt.target._name);
};
box_ldr.addEventListener("progress", ldrListener);
box_ldr.addEventListener("complete", ldrListener);
box_ldr.load("http://www.macromedia.com/images/shared/product_boxes/159x120/15
  9x120_box_flashpro.jpg");
```

The following example shows another recommended way of using a listener. In this example, you define a function that is called when the user presses a Button component instance on the Stage:

```
var submit_button:mx.controls.Button;
submit_button.clickHandler = function(evt:Object) {
  trace(evt.target._name);
}
```

By appending `Handler` to the event name (in this case, `click`), the event is caught, and you trace an instance name when a user clicks the button.

A slight variation on the first example in this section is to use the `handleEvent` method, but this technique is slightly more cumbersome. It is not recommended, because you must use a series of `if..else` statements or a `switch` statement to detect what event is caught.

```
var box_ldr:mx.controls.Loader;
var ldrListener:Object = new Object();

ldrListener.handleEvent = function(evt:Object) {
  switch (evt.type) {
  case 'progress' :
    trace("loader loading:"+Math.round(evt.target.percentLoaded)+"%");
    break;
  case 'complete' :
    trace("loader complete:"+evt.target._name);
    break;
  }
};
box_ldr.addEventListener("progress", ldrListener);
box_ldr.addEventListener("complete", ldrListener);
box_ldr.load("http://www.macromedia.com/images/shared/product_boxes/159x120/15
  9x120_box_flashpro.jpg");
```

Following general formatting guidelines

Adding spacing (or white space) to your syntax is recommended because it makes your ActionScript easier to read. The following formatting points are recommended to help promote readability in your ActionScript.

The following example includes a space after a keyword that is followed by parentheses [()]:

```
do {
  //something
} while (condition);
```

You do not have to put a space between a method name and a following parentheses, as the following example shows:

```
function checkLogin() {
  //statements;
}
checkLogin();
```

If you follow these guidelines, it is easier to distinguish between method calls and keywords.

In a list of arguments, such as in the following example, include a space after commas:

```
function addItems(item1:Number, item2:Number):Number {
  return (item1+item2);
}
var sum:Number = addItems(1, 3);
```

Separate all operators and their operands by spaces, as the following ActionScript shows:

```
var sum:Number = 7 + 3;
```

An exception to this guideline is the dot (`.`) operator. Unary operators such as increment (++) and decrement (--) do not have spaces.

Using scope

Scope is the area where the variable is known and can be used in a SWF file, such as on the timeline, globally across an application, or locally within a function. There are three kinds of variables: local, timeline, and global. For information on these variables and scope, see "Scoping and declaring variables" on page 45. ActionScript 2.0 classes also support public, private, and static variable scopes.

It is important to understand the difference between the `_global` and `_root` scopes. The `_root` scope is unique for each loaded SWF file. Use the `_global` identifier to create global objects, classes, or variables. The global scope applies to all Timelines and scopes within SWF files. Use relative addressing rather than references to `_root` timelines, because it makes code reusable and portable. See the following sections for the recommended way of scoping variables. For more information on scope, see "Event handler scope" on page 174 and "Scope of the this keyword" on page 176. For information on using the `super` prefix, see "Using the super prefix" on page 87.

For more information on scope, see the following sections:

- "Avoiding _root" on page 96
- "Using _lockroot" on page 96

Avoiding _root

There are several ways to target instances that let you avoid using `_root`; these are discussed later in this section. Avoid using `_root` in ActionScript because it can cause SWF files that load into other SWF files to not work correctly. The `_root` identifier targets the base SWF file that is loading, not the SWF file using relative addressing instead of `_root`. This issue limits code portability in SWF files that are loaded into another file, and, particularly, in components and movie clips. You can help resolve problems using `_lockroot`, but only use `_lockroot` when necessary (such as when you are loading a SWF file but do not have access to the FLA file). For more information on using `_lockroot`, see "Using _lockroot" on page 96.

Use `this`, `this._parent`, or `_parent` keywords rather than `_root`, depending on where your ActionScript is located. The following example shows relative addressing:

```
myClip_mc.onRelease = function() {
  trace(this._parent.myButton_btn._x);
};
```

All variables must be scoped, except for variables that are function parameters and local variables. Scope variables relative to their current path whenever possible, using relative addressing, such as the `this` keyword. For more information on using the `this` keyword, see `this` in *Flash ActionScript Language Reference*.

Using _lockroot

Flash MX 2004 introduced `_lockroot` as a way to solve the scoping issues sometimes associated with using `_root`. Although this solves many problems with applications, consider `_lockroot` as a workaround for problems caused by using `_root`. If you experience problems loading content into a SWF file or a component instance, try applying `_lockroot` to a movie clip that loads the content. For example, if you have a movie clip called `myClip_mc` loading content, and it ceases working after it is loaded, try using the following code (placed on the main Timeline):

```
this._lockroot = true;
```

For more information on using `_lockroot`, see `MovieClip._lockroot` in *Flash ActionScript Language Reference*.

Using the this keyword

Whenever possible, use the `this` keyword as a prefix instead of omitting the keyword, even if your code works without it. You can use the `this` keyword to learn when a method or property belongs to a particular class. For example, for a function on the main Timeline, write ActionScript using the following format:

```
circle_mc.onPress = function() {
  this.startDrag();
};
circle_mc.onRelease = function() {
```

```
  this.stopDrag();
};
```

For a class, you can write code in the following format:

```
class User {
  private var m_username:String;
  private var m_password:String;
  function User(username:String, password:String) {
    this.m_username = username;
    this.m_password = password;
  }
  public function get username():String {
    return this.m_username;
  }
  public function set username(username:String):Void {
    this.m_username = username;
  }
}
```

If you consistently add the `this` keyword in these situations, your ActionScript will be much easier to read and understand.

Using scope in classes

When you port code to ActionScript 2.0 classes, you might have to change how you use the `this` keyword. For example, if you have a class method that uses a callback function (such as the LoadVars class's `onLoad` method), it can be difficult to know if the `this` keyword refers to the class or the LoadVars object. In this situation, it might be necessary to create a pointer to the current class, as the following example shows:

```
class Product {
  private var m_products_xml:XML;
  // constructor
  // targetXml_string - contains the path to an XML file
  function Product(targetXml_string:String) {
    /* Create a local reference to the current class.
      Even if you are within the XML's onLoad event handler, you
      can reference the current class instead of only the XML packet. */
    var thisObj:Product = this;
    // Create a local variable, which is used to load the XML file.
    var prod_xml:XML = new XML();
    prod_xml.ignoreWhite = true;
    prod_xml.onLoad = function(success:Boolean) {
      if (success) {
        /* If the XML successfully loads and parses,
          set the class's m_products_xml variable to the parsed
          XML document and call the init function. */
        thisObj.m_products_xml = this;
        thisObj.init();
      } else {
        /* There was an error loading the XML file. */
        trace("error loading XML");
      }
    };
```

```
    // Begin loading the XML document.
    prod_xml.load(targetXml_string);
  }
  public function init():Void {
    // Display the XML packet.
    trace(this.m_products_xml);
  }
}
```

Because you are trying to reference the private member variable within an `onLoad` handler, the `this` keyword actually refers to the `prod_xml` instance and not the `Product` class, which you might expect. For this reason, you must create a pointer to the local class file so that you can directly reference the class from the `onLoad` handler.

Using functions

Reuse blocks of code whenever possible. One way you can reuse code is by calling a function multiple times, instead of creating different code each time. Functions can be generic pieces of code, so you can use the same blocks of code for slightly different purposes in a SWF file. Reusing code lets you create efficient applications and minimize the ActionScript that you must write, which reduces development time. You can create functions in a class file or write ActionScript that resides in a code-based component.

If you are using ActionScript 2.0, do not place functions on the Timeline. When using ActionScript 2.0, place functions into class files whenever possible, as the following example shows:

```
class Circle {
public function area(radius:Number):Number {
  return (Math.PI*Math.pow(radius, 2));
}
public function perimeter(radius:Number):Number {
  return (2 * Math.PI * radius);
}
public function diameter(radius:Number):Number {
  return (radius * 2);
}
}
```

Use the following syntax when you create functions:

```
function myCircle(radius:Number):Number {
  //...
}
```

Avoid using the following syntax, which is difficult to read:

```
myCircle = function(radius:Number):Number {
  //...
}
```

The following example puts functions into a class file. This is a best practice when you choose to use ActionScript 2.0, because it maximizes code reusability. When you want to reuse the functions in other applications, you can import the existing class rather than rewrite the code from scratch, or duplicate the functions in the new application.

```
class mx.site.Utils {
  static function randomRange(min:Number, max:Number):Number {
    if (min>max) {
      var temp:Number = min;
      min = max;
      max = temp;
    }
    return (Math.floor(Math.random()*(max-min+1))+min);
  }
  static function arrayMin(num_array:Array):Number {
    if (num_array.length == 0) {
      return Number.NaN;
    }
    num_array.sort(Array.NUMERIC | Array.DESCENDING);
    var min:Number = Number(num_array.pop());
    return min;
  }
  static function arrayMax(num_array:Array):Number {
    if (num_array.length == 0) {
      return undefined;
    }
    num_array.sort(Array.NUMERIC);
    var max:Number = Number(num_array.pop());
    return max;
  }
}
```

You might use these functions by adding the following ActionScript to your FLA file:

```
import mx.site.Utils;
var randomMonth:Number = Utils.randomRange(0, 11);
var min:Number = Utils.arrayMin([3, 3, 5, 34, 2, 1, 1, -3]);
var max:Number = Utils.arrayMax([3, 3, 5, 34, 2, 1, 1, -3]);
trace("month: "+randomMonth);
trace("min: "+min);
trace("max: "+max);
```

Using classes and ActionScript 2.0

You create classes in separate ActionScript files that are imported into a SWF file when it is compiled. To create a class file, the code you write can have a certain methodology and ordering, which is discussed in the following sections.

Use the following basic ordering in your AS class files:

1. Documentation comments
2. Package and import statements
3. Class and interface declarations

The Flash MX 2004 object model does not support multiple inheritance; it supports only single inheritance. Therefore, a class can come from a single parent class. This parent class can be either a native Flash class or a user-defined class. Flash MX 2004 uses interfaces to provide additional functionality, but only supports single inheritance. For information on interfaces, see "Interfaces" on page 249. For information on implementing inheritance, see "Inheritance" on page 249. For more information on Documentation comments, see "Using comments in classes" on page 102. For information on import statements and class declarations, see "Creating a class file" on page 251.

In ActionScript, the distinction between interface and object is only for the compile-time error checking and language rule enforcement. An interface is not a class; however, this is not altogether true in ActionScript at runtime. An interface is abstract, similar to the way it is in Java. However, ActionScript interfaces do exist at runtime to allow type casting. An interface is not an object or a class.

For more information, see the following topics:

- "Creating and organizing classes" on page 100
- "Programming classes" on page 102
- "Using prefixes in classes" on page 102
- "Using comments in classes" on page 102
- "Wrapping lines of code" on page 103
- "Using design patterns" on page 104

Creating and organizing classes

This section describes the structure of a class file. The following guidelines show how parts of a class are ordered to increase efficiency and improve the readability in your code. For more information on naming classes and parts of a class, see "General naming guidelines" on page 69.

The class file begins with documentation comments that include a general description of the code, in addition to author information and version information.

Import statements follow the documentation comments. The first part of a package name is written in lowercase letters and follows an established naming convention. For example, if you are following the recommended naming guidelines, you might use the following statement:

```
import com.macromedia.Utils;
```

The next line of a class is a package statement, class declaration, or interface declaration, as the following example shows:

```
class com.macromedia.users.UserClass{...}
```

Directly following the statement or declaration, include any necessary class or interface implementation comments. Add information in this comment that is pertinent for the entire class or interface.

Following the class or interface implementation comments, add all your static variables. Write the public class variables first and follow them with private class variables.

Instance variables follow static variables. Write the public member variables first, and follow them with private member variables.

Following the public and private member variables, add the constructor statement, such as the one in the following example:

```
public function UserClass(username:String, password:String) {...}
```

Finally, write your methods. Group methods by their functionality, not by their accessibility or scope. Organizing methods this way helps improve the readability and clarity of your code. Then write the getter/setter methods into the class file.

In general, only place one declaration per line, and do not place either the same or different types of declarations on a single line. Format your declarations as the following example shows:

```
var prodSKU:Number;    // product SKU (identifying) number
var prodQuantity:Number;  // quantity of product
```

This example shows better form than putting both declarations on a single line. Place these declarations at the beginning of a block of code enclosed by braces (`{}`).

Initialize local variables when they are declared, unless that initial value is determined by a calculation. Declare variables before you first use them, except in `for` loops.

Avoid using local declarations that hide higher level declarations. For example, do not declare a variable twice, as the following example shows:

```
var counter:Number = 0;
function myMethod() {
  for (var counter = 0; counter<=4; counter++) {
    //statements;
  }
}
```

This code declares the same variable inside an inner block, which is a practice you should avoid.

The following example shows the organization of a simple class:

```
class com.macromedia.users.UserClass {
  private var m_username:String;
  private var m_password:String;
  public function UserClass(username:String, password:String) {
    this.m_username = username;
    this.m_password = password;
  }
  public function get username():String {
    return this.m_username;
  }
  public function set username(username:String):Void {
    this.m_username = username;
  }
  public function get password():String {
    return this.m_password;
  }
  public function set password(password:String):Void {
    this.m_password = password;
  }
}
```

Programming classes

There are several general guidelines for programming classes. These guidelines help you write well-formed code, but also remember to follow the guidelines provided in "General coding conventions" on page 69 and "ActionScript coding standards" on page 82. When you program classes, follow these guidelines:

- Do not use objects to access static methods and variables. Do not use `myObj.classMethod();` use a class name, such as `MyClass.classMethod()`.
- Do not assign many variables to a single value in a statement, because it is difficult to read, as the following ActionScript shows:

```
play_btn.onRelease = play_btn.onRollOut = playsound;
```

  or:

```
class User {
  private var m_username:String, m_password:String;
}
```

- Have a good reason for making public instance or public static, class, or member variables. Make sure that these variables are explicitly public before you create them this way.
- Do not use too many getter/setter functions in your class files, and access them frequently in your application.

Using prefixes in classes

Whenever possible, use the `this` keyword as a prefix within your classes for methods and member variables. It is easy to tell that a property or method belongs to a class when it has a prefix; without it, you cannot tell if the property or method belongs to the superclass.

For an example of using prefixes in classes, see the UserClass in "Creating and organizing classes" on page 100.

You can also use a class name prefix for static variables and methods, even within a class. This helps qualify the references you make, which makes code readable. Depending on what coding environment you are using, using prefixes might also trigger code completion and hinting.

You do not have to add these prefixes, and some developers feel it is unnecessary. Adding the `this` keyword as a prefix is recommended, because it can aid readability and helps you write clean code by providing context.

Using comments in classes

Using comments in your classes and interfaces is an important part of documenting them. Start all your class files with a comment that provides the class name, its version number, the date, and your copyright. For example, you might create documentation for your class that is similar to the following comment:

```
/**
  User class
  version 1.2
  3/21/2004
  copyright Macromedia, Inc.
 */
```

There are two kinds of comments in a typical class or interface file: *documentation comments* and *implementation comments*. Documentation comments are used to describe the code's specifications and do not describe the implementation. Implementation comments are used to comment out code or to comment on the implementation of particular sections of code. The two kinds of comments use slightly different delimiters. Documentation comments are delimited with /** and */, and implementation comments are delimited with /* and */. For more information on why comments are included in code, see "Using comments in code" on page 77.

Documentation comments are used to describe interfaces, classes, methods, and constructors. Include one documentation comment per class, interface, or member, and place it directly before the declaration. If you have additional information to document that does not fit into the documentation comments, use implementation comments (in the format of block comments or single-line comments, described next). Implementation comments directly follow the declaration.

Note: Do not include comments that do not directly relate to the class being read. For example, do not include comments that describe the corresponding package.

Block comments These comments describe files, data structures, methods, and descriptions of files. Place a blank line before a block comment. They are usually placed at the beginning of a file and before or within a method. The following ActionScript is an example of a block comment.

```
/*
  Block comment
*/
```

Single-line comments These comments are typically used to explain a small code snippet. You can use single-line comments for any short comments that fit on a single line. The following example includes a single-line comment:

```
while (condition) {
  // handle condition with statements
}
```

Trailing comments These comments appear on the same line as your ActionScript code. Space the comments to the right, so they can be distinguished from the code. Try to have the comments line up with each other, if possible, as the following code shows:

```
var myAge:Number = 27;            //my age
var myCountry:String = "Canada";  //my country
var myCoffee:String = "black";    //my coffee preference
```

For more information on spacing and formatting, see "Spacing and readability" on page 81.

Wrapping lines of code

Sometimes your expressions do not fit on a single line. Using word wrap in the Script pane or ActionScript editor solves this problem, but sometimes you have to break expressions, particularly when code is printed on a page or in an electronic document. Use the following guidelines when breaking lines of code:

- Break a line before an operator.
- Break a line after a comma.
- Align the second line with the start of the expression on the previous line of code.

Using design patterns

Design patterns help developers structure their application in a particular, established way. There are many different design patterns that developers use in classes and for application design. Using a design pattern is helpful when working in larger groups, because there is a defined set of guidelines. Design patterns help ensure that every developer in the group can read a snippet of code and understand what is happening. The guidelines keep the code layout, architecture, placement, and style consistent throughout the project, regardless of who writes the code. Design patterns might also make developing applications more efficient, because you can reuse the ActionScript that you write in several different user interfaces.

A common use of class members is the *Singleton design pattern*. The Singleton design pattern makes sure that a class has only one instance, and provides a way of globally accessing the instance. For more information on the Singleton design pattern, see www.macromedia.com/devnet/mx/coldfusion/articles/design_patterns.html.

Often there are situations when you need exactly one object of a particular type in a system. For example, in a chess game there is only one chessboard, and in a country, there is only one capital city. Even though there is only one object, it is attractive to encapsulate the functionality of this object in a class. However, you might need to manage and access the one instance of that object. Using a global variable is one way to do this, but global variables are often not desirable. A better approach is to make the class manage the single instance of the object itself using class members, such as the following:

```
class Singleton {
  private var instance:Singleton = null;
  public function doSomething():Void {
    //...
  }
  public static function getInstance():Singleton {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

The Singleton object can then be accessed using `Singleton.getInstance();` This also means that the Singleton object is not created until it is actually needed—that is, until some other code asks for it by calling the `getInstance` method. This is typically referred to as *lazy creation*, and can help code efficiency in many circumstances.

Note: Remember to not use too many classes for your application, because it can create many poorly designed class files, which is not beneficial to the application's performance or your workflow.

Behaviors conventions

Behaviors are prewritten code snippets that can be instantly added to parts of a FLA file. The introduction of behaviors has added to the complexity of determining best practices in Flash, because the way some behaviors are added does not follow typical and ideal workflows. Many developers usually enter ActionScript either into one or several frames on the main Timeline or in external ActionScript files, which is a good practice to follow. However, when you use behaviors, sometimes code is placed directly on symbol instances (such as buttons, movie clips, or components) instead of being placed on the Timeline.

Behaviors are convenient, save substantial time, and can be useful for novice Flash and ActionScript users. Before you start using behaviors, take a close look at how you want to structure your FLA file:

- What behaviors do you need for your project?
- What code do the behaviors contain?
- How are you are going to use and implement behaviors?
- What other ActionScript do you need to add?

If you carefully plan a document that uses behaviors, you can avoid problems that could be created by decentralizing your ActionScript.

For more information, see the following topics:

- "Comparing timeline code with object code" on page 105
- "Using behaviors" on page 106
- "Being consistent" on page 107
- "Being courteous" on page 107

Comparing timeline code with object code

Planning a project and organizing a document or application cannot be underestimated, particularly when you are creating large involved projects or working in teams. This is why the placement of ActionScript—often what makes the project work—is important.

Many developers do not place ActionScript on symbol instances, and instead place their code on the Timeline (timeline code) or in classes. Because Behaviors add code to many locations in a FLA file, it means that your ActionScript is not centralized and can be difficult to locate. When code is not centralized, it is difficult to figure out interactions between the snippets of code, and it is impossible to write code in an elegant way. It can potentially lead to problems debugging code or editing files. Many developers also avoid placing code on different frames on the Timeline or avoid placing timeline code inside multiple movie clips where it is hidden. By placing all your code, including functions that must be defined before they are used, in a SWF file, you can avoid such problems.

Flash has features that make it easy to work with behaviors in a document and with decentralized ActionScript. If you use behaviors, try the following features when working on your project:

Script navigator Makes your timeline code or code on individual objects easy to find and edit in the Actions panel.

Find and replace Lets you search for strings and replace them in a FLA document.

Script pinning Lets you pin multiple scripts from various objects and work with them simultaneously in the Actions panel. This works best with the Script navigator.

Movie Explorer Lets you view and organize the contents of a FLA file, and select elements (including scripts) for further modification.

Using behaviors

Knowing when to use behaviors is the most important guideline. Carefully consider your project and whether behaviors are the best solution for you, which can be determined by answering the questions that follow. Consider different ways of structuring your projects, as well as the different options and features available in Flash.

If you have a FLA file with symbols, you can select one of the instances on the Stage, and then use the Add menu on the Behaviors panel to add a behavior to that instance. The behavior you select automatically adds code that attaches to the instance, using code such as the `on()` handler. You can also select a frame on the Timeline, or a slide or form in a screen-based FLA file, and add different behaviors to a frame or screen using the Behaviors panel.

You need to decide when you need to use behaviors instead of writing ActionScript. First, answer the questions in the introductory section "Behaviors conventions" on page 105. Examine how and where you want to use behaviors and ActionScript in your FLA file. Then, consider the following questions:

- Do you have to modify the behavior code? If so, by how much?
- Do you have to interact with the behavior code with other ActionScript?
- How many behaviors do you have to use, and where do you plan to put them in the FLA file?

Your answers to these questions determine whether you should use behaviors. If you want to modify the behavior code to any extent, do not use behaviors. Behaviors usually cannot be edited using the Behaviors panel if you make modifications to the ActionScript. And if you plan to significantly edit the behaviors in the Actions panel, it is usually easier to write all of the ActionScript yourself in a centralized location. Debugging and modifications are easier to make from a central location than having code generated by behaviors placed in many areas around your FLA file. Debugging and interaction can be inelegant or difficult with scattered code, and sometimes it is easier to write the ActionScript yourself.

The main difference between a FLA file with behaviors and a FLA file without behaviors is the workflow you must use for editing the project. If you use behaviors, you must select each instance on the Stage, or select the Stage, and open the Actions or Behaviors panel to make modifications. If you write your own ActionScript and put all your code on the main Timeline, you only have to go to the Timeline to make your changes.

Use behaviors consistently throughout a document when they are your main or only source of ActionScript. It is best to use behaviors when you have little or no additional code in the FLA file, or have a consistent system in place for managing the behaviors that you use.

Being consistent

There are some guidelines for using behaviors; the main thing is consistency. If you add ActionScript to a FLA file, put code in the same locations where behaviors are added, and document how and where you add code.

Note: If you are using a screen-based FLA file, see "Screens conventions" on page 107 for more information on best practices and screens.

For example, if you place code on instances on the Stage, on the main Timeline, and in class files, you should examine your file structure. Your project will be difficult to manage, because the code placement is inconsistent. However, if you logically use behaviors and structure your code to work in a particular way surrounding those behaviors (place everything on object instances), your workflow is logical and consistent. The document will be easier to modify later.

Being courteous

If you plan to share your FLA file with other users and you use ActionScript placed on or inside objects (such as movie clips), it can be difficult for those users to find your code's location, even when they use the Movie Explorer to search through the document.

If you are creating a FLA file that has *spaghetti code* (code placed in many locations throughout the document) and plan to share the file, it is courteous to notify other users that you are using ActionScript that is placed in or on objects. This courtesy ensures that other users immediately understand the structure of the file. Leave a comment on Frame 1 on the main Timeline to tell users where to find the code and how the file is structured. The following example shows a comment that tells users the location of the ActionScript:

```
/*
  On Frame 1 of main Timeline.
  ActionScript placed on component instances and inside movie clips using
   behaviors.
  Use Movie Explorer to locate ActionScript
*/
```

Note: It is not necessary to use this technique if your code is easy to find, the document is not shared, or all of your code is placed on frames of the main Timeline.

Clearly document the use of behaviors if you are working with a complex document. If you keep track of where you use behaviors, you might have fewer headaches in the long run. Perhaps you can create a flow chart or list, or use good documentation comments in a central location on the main Timeline.

Screens conventions

Screens introduce a new way to develop applications by organizing assets, which can dramatically reduce the time to write an application. You can use screens with or without using the Timeline. The process that's used to organize documents might seem logical to some developers, or make more sense for certain screen-based projects; for example, if you have to create an application that follows a linear process or has multiple states, such as one that requires server validation or multipart forms that a user must fill out and send to a database. You can also use classes that are built into screens to quickly and easily add additional functionality to your application.

Like the Behaviors guidelines, there are issues with how to organize and structure projects built with the screen-based authoring environment. Screens provide an intelligent and easy to use framework to control loading, persistence of data, and state using classes.

Some developers build applications with all their ActionScript in a centralized location. Other designers and developers, usually newer to Flash, might use a more visual approach to writing a screens document. Code placement is a central issue with screens and is discussed in this section.

For more information, see the following topics:

- "Organizing code for screens" on page 108
- "Working with other structural elements" on page 110

Organizing code for screens

There are three places you can place code in a screen-based application:

- On the Timeline
- On screens and symbol instances
- In an external file

Because code can be placed in many different locations, it complicates matters as to where you should put your code. Therefore, you must consider the type of application you're writing and what it requires in the way of ActionScript. As with behaviors, you should use ActionScript consistently in screen-based applications.

The difference between screens and behaviors is that the ActionScript that behaviors add is much more complex than most of the behaviors available for a regular FLA file. Screens are based on complex ActionScript, so some of the code used for transitions and changing slides might be difficult to write yourself.

You might use either behaviors or ActionScript that attaches directly to screens, combined with either a Timeline or an external ActionScript file. Even if you decentralize your code this way, have code put on screens and an external ActionScript file, you should still avoid attaching code directly to movie clip or button instances that are placed on individual screens. This ActionScript is still hard to locate in a FLA file, debug, and edit.

Even if you attach code directly to a screen, it is more acceptable and easier to use than in regular FLA files for the following reasons:

- The code that attaches to screens when you use behaviors often doesn't interact with other ActionScript you might write—you can place behaviors there and you might not have to worry about editing the code further, which is ideal.
- The code placed directly on screens is easy to locate and view the hierarchy of, because of the Screen Outline pane. Therefore, it is easy to quickly locate and select all of the objects that you might have attached ActionScript to.

If you use behaviors placed on screens (or other instances), remember to document the location on Frame 1 of the main Timeline. This is particularly important if you also place ActionScript on the Timeline. The following code is an example of the comment you might want to add to your FLA file:

```
/*
  On Frame 1 of main Timeline.
  ActionScript is placed on individual screens and directly on instances in
  addition to the code on the Timeline (frame 1 of root screen).
  ...
*/
```

Placing code in the FLA file

Using behaviors on screens while placing ActionScript on the main Timeline makes a screen-based FLA file less complex and easier to work with than a regular FLA document. Behavior code is sometimes added to instances where it might take a long time to create because of its complexity. The convenience of using behaviors might vastly outweigh any drawbacks if the behaviors you add to a screens document are quite complex to write yourself.

New Flash users frequently like the visual approach of placing ActionScript for a particular screen directly on an object. When you click the screen or a movie clip, you see the code that corresponds to the instance or the name of the function that's called for that instance. This makes navigating an application and associated ActionScript visual. It's also easier to understand the hierarchy of the application while in the authoring environment.

If you decide to attach ActionScript to symbol instances on the Stage and directly on screens, try to place all your ActionScript only in these two places to reduce complexity.

If you place ActionScript on screens and either on the Timeline or in external files, try to place all your ActionScript in only these two of places to reduce complexity.

Using external ActionScript

You can organize your screen-based FLA file by writing external code and not having any code in the document. When you use external ActionScript, try to keep most of it in external AS files to avoid complexity. Placing ActionScript directly on screens is acceptable, but avoid placing ActionScript on instances on the Stage.

You can create a class that extends the Form class. For example, you could write a class called `MyForm`. In the Property inspector, you would change the class name from `mx.screens.Form` to `MyForm`. The `MyForm` class would look similar to the following code:

```
class MyForm extends mx.screens.Form {
  function MyForm() {
    trace("constructor: "+this);
  }
}
```

Working with other structural elements

A screen-based document, when published, is essentially a single movie clip on the first frame of a Timeline. This movie clip contains a few classes that compile into the SWF file. These classes add additional file size to the published SWF file compared with a nonscreen-based SWF file. The contents load into this first frame by default, which might cause problems in some applications.

You can load content into a screen-based document as separate SWF files onto each screen to reduce the initial loading time. Load content when it is needed, and use runtime shared libraries when possible. This approach reduces what the user needs to download from the server, which reduces the time that the user must wait for content if they do not have to view each different part of the application.

Video conventions

The use of video in Flash has greatly increased and improved from earlier versions of Flash. There are many options to make edits to video before you import footage into a FLA document. There are also greater controls for video compression when you import it into Flash. Compressing video carefully is important because it controls the quality of the footage and the size of the file. Video files, even when compressed, are large in comparison with most other assets in your SWF file.

Note: Remember to provide the user with control over the media in a SWF file. For example, if you add audio to a document with video (or even a looping background sound), let the user control the sound.

For more information, see the following topics:

- "Using video" on page 110
- "Importing and embedding video" on page 111
- "Importing and embedding video" on page 111
- "Using Media components" on page 113
- "Dynamically loading video using ActionScript" on page 113

Using video

Before you import video into Flash, consider what video quality you need, what video format you want to use with the FLA file, and how you want it to download. You can import the footage directly into a SWF file using any video file format that is supported by Microsoft Direct Show or Apple QuickTime. (Formats include AVI, MPG, MPEG, MOV, DV, WMV and ASF.)

When you import video into a FLA file, it increases the size of the SWF file that you publish. This video starts downloading to the user's computer whether or not they view the video. You can also stream the video from an external Flash Video (FLV) file on your server.

Note: Video progressively downloads from the server like SWF files, which is not actually *streaming*. Even dynamically loading content has distinct advantages over keeping all your content in a single SWF file. For example, you will have smaller files and quicker loading, and the user only downloads what they want to see or use in your application.

Give users a certain amount of control (such as the ability to stop, pause, play, and resume the video, and control volume) over the video in a SWF file. For more information on using video in Flash, see "Working with Video" in *Using Flash*.

Importing and embedding video

You can embed video in a SWF file by importing it into your FLA document. You can import the video directly into the library, where it is stored as an *embedded video*. You can also import the footage directly onto the main Timeline or into a movie clip. When you place video on the Timeline by importing or dragging it from the library, a dialog box appears prompting you to extend the current Timeline by a specified number of frames.

You might need flexibility over your video, such as manipulating it or syncing various parts of it with the Timeline. If you need flexibility or advanced control, you should embed your video in the SWF file rather than loading it using ActionScript or one of the Media components.

When you work with embedded video, a best practice is to place video inside a movie clip instance, because you have the most control over the content. The video's Timeline plays independently from the main Timeline. You do not have to extend your main Timeline by many frames to accommodate for the video, which can make working with your FLA file difficult.

To import video:

1. Select File > Import > Import to Library.
2. Select the video footage that you want to import.
3. Step through the Video Import wizard to edit, compress, and embed the video.

The video is placed in the library. This process is the recommended way to import video, because it gives you the most control over how you work with the video and where you place it in the FLA file.

Exporting FLV files

You can export FLV files from Flash MX 2004 and Flash MX Professional 2004 authoring environments. After you import video into your document, it appears as a video symbol in the library.

FLV files use the FLV mime type `video/x-flv`. If you have difficulty viewing FLV files after you upload your files, check that this mime type is set on your server. FLV files are binary, and some applications that you build might require that the `application/octet-stream` subtype is also set. For more information on the Flash Player specifications, see http://download.macromedia.com/pub/flash/flash_file_format_specification.pdf.

To export video as an FLV file:

1. Select the video symbol in the library, right-click (Windows) or Control-click (Macintosh), and select Properties from the context menu.
2. Click Export in the Embedded Video Properties dialog box to open the Export FLV dialog box.
3. Enter a name for the file, and select a location to save it.
4. Click Save, and the video is exported as an FLV file.

Note: Remember to delete the FLV file from your Flash document and library if you intend to dynamically load the video into that document at runtime.

Flash MX Professional includes an external FLV Exporter that compresses video from third-party video editing software such as QuickTime Pro and Adobe After Effects. The quality of the FLV file that is created using this tool is better than video exported directly from Flash.

When you compress video, remember the following recommendations:

Do not recompress video Recompressing video leads to quality degradation, such as artifacts. Try to use raw footage or the least compressed footage that is available to you.

Make your video as short as possible Trim the beginning and end of your video so it is as short as possible, and edit your video to remove any unnecessary content. This can be accomplished directly in Flash using the Video Import wizard.

Adjust your compression settings If you compress footage and it looks great, try changing your settings to reduce the file size. Test your footage, and modify it until you find the best setting possible for the video you are compressing. Remember that all video has varying attributes that affect compression and file size; each video needs its own setting for the best results.

Limit effects and rapid movement Limit movement as much as possible if you are concerned about file size. Any kind of movement, particularly with many colors, increases file size. For example, effects (such as cross fades, blurs, and so on) increase file size, because the video contains more information.

Choose appropriate dimensions If your target audience has a slow Internet connection (such as phone modems), you should make the dimensions of your video smaller, such as 160x120 pixels. If your visitors have fast connections, you can make your dimensions larger (for example, 320x240 pixels).

Choose appropriate frames per second Choose an appropriate number of frames per second (*fps*). If your target audience has slow Internet connections (such as phone modems), you should choose a low rate of frames per second (such as 7 or 15 fps). If your visitors have fast connections, you can use a higher rate of frames per second (such as 15 or 30 fps). You should always choose a frames per second that is a multiple of your original frame rate. For example, if your original frame rate was 30 fps, you should compress to 15 fps or 7.5 fps.

Choose an appropriate number of keyframes Video keyframes are different from keyframes in Flash. Each keyframe is a frame that draws when the video is compressed, so the more frequent your keyframes are the better quality the footage will be. More keyframes also mean a higher file size. If you choose 30, a video keyframe draws every 30 frames. If you choose 15, the quality is higher because a keyframe draws ever 15 frames and the pixels in your footage are more accurate to the original.

Reduce noise Noise (scattered pixels in your footage) increases file size. Try reducing noise using your video editor, to reduce the video file size. Using more solid colors in your video reduces its file size.

Using Media components

Media components are used to display FLV files or play MP3 files in a SWF file, and they only support these two file types. You can export each file format using a variety of software. For information on exporting the FLV format, see "Exporting FLV files" on page 111.

Media components are easy to use, so you do not have to write ActionScript to dynamically load FLV files. The components are available only in Flash MX Professional 2004, but you can use them to dynamically load FLV files into a SWF file.

Note: MP3 and FLV files progressively download into a SWF file. Use Flash Communication Server to stream media to a SWF file.

There are several ways that you can load video into a SWF file using Media components. The following procedure is the basic recommended way to use Media components; however, there are many additional settings you can make.

To use Media components:

1. Drag a MediaPlayback or MediaDisplay component onto the Stage.
2. Select the component instance, and use the Component inspector panel to enter the file format and location of the media that you want to dynamically load into your SWF file.
3. Select how you want your video controls to appear.
4. Use the Component inspector panel to set the location of the controls and whether they are visible, hidden, or minimized during video playback.
5. Enter the length of the FLV file for the playhead to recognize the length of the video and follow the video as it plays.

   Note: See the following example to determine the length of an FLV file using ActionScript.

Note: You can enter additional properties and cue points using the Component inspector panel.

The recommended way to find out the length of an FLV file is to use code in the following format:

```
var listenerObject:Object = new Object();
listenerObject.complete = function(evt:Object) {
  trace("seconds: "+evt.target.playheadTime);
};
myMedia.addEventListener("complete", listenerObject);
```

After the FLV file finishes playing in the Media component called `myMedia`, it displays the total number of seconds of the video in the Output panel.

Dynamically loading video using ActionScript

You do not have to use Media components to dynamically load FLV files into a SWF file. You can use ActionScript with a Video object instance to load the video into a SWF file at runtime.

To dynamically load video using ActionScript:

1. Add a video object to the Stage by selecting New Video from the Library panel's options menu.
2. Drag the video object on to the Stage, and resize the instance using the Property inspector to the same dimensions as your FLV file.
3. Enter an instance name of `video1_video` in the Property inspector.
4. Select Frame 1 of the Timeline, and enter the following ActionScript into the Actions panel:

```
var connection_nc:NetConnection = new NetConnection();
connection_nc.connect(null);
var stream_ns:NetStream = new NetStream(connection_nc);
stream_ns.setBufferTime(3);
video1_video.attachVideo(stream_ns);
stream_ns.play("video1.flv");
```

5. Rename `video1.flv` in the ActionScript to the name of the FLV file that you want to load.
6. Save the FLA document in the same directory as your FLV file.

Performance and Flash Player

SWF file performance is important, and you can improve performance in many ways: from how you write ActionScript, to how you build animations. This section provides guidelines and practices that help improve the performance of your SWF file at runtime.

For more information, see the following topics:

- "Optimizing graphics and animation" on page 114
- "Working with components in Flash Player" on page 115
- "Preloading components and classes" on page 117
- "Working with text" on page 118
- "Optimizing ActionScript in Flash Player" on page 120

Optimizing graphics and animation

The first step in creating optimized and streamlined animations or graphics is to outline and plan your project before its creation. Make a target for the file size and length of the animation that you want to create, and test throughout the development process to ensure that you are on track. If you are creating advertisements, for example, length and file size are extremely important.

Avoid using gradients, because they require many colors and calculations to be processed, which is more difficult for a computer processor to render. For the same reason, keep the amount of alpha or transparency you use in a SWF file to a minimum. Animating objects that include transparency is processor-intensive and should be kept to a minimum. Animating transparent graphics over bitmaps is a particularly processor-intensive kind of animation, and must be kept to a minimum or avoided completely.

Note: The best bitmap format to import into Flash is PNG, which is the native file format of Macromedia Fireworks. PNG files have RGB and alpha information for each pixel. If you import a Fireworks PNG file into Flash, you retain some ability to edit the graphic objects in the FLA file.

Optimize bitmaps as much as possible without overcompressing them. A 72-dpi resolution is optimal for the web. Compressing a bitmap image reduces file size, but compressing it too much compromises the quality of the graphic. Check that the settings for JPEG quality in the Publish Settings dialog box do not overcompress the image. If your image can be represented as a vector graphic, this is preferable in most cases. Using vector images reduces file size, because the images are made from calculations instead of many pixels. Limit the number of colors in your image as much as possible while still retaining quality.

Note: Avoid scaling bitmaps larger than their original dimensions, because it reduces the quality of your image and is processor-intensive.

Set the `_visible` property to `false` instead of changing the `_alpha` level to 0 or 1 in a SWF file. Calculating the `_alpha` level for an instance on the Stage is processor-intensive. If you disable the instance's visibility, it saves CPU cycles and memory, which can give your SWF files smoother animations. Instead of unloading and possibly reloading assets, set the `_visible` property to `false`, which is much less processor-intensive.

Try to reduce the number of lines and points you use in a SWF file. Use the Optimize Curves dialog box (Modify > Shape > Optimize) to reduce the number of vectors in a drawing. Select the Use Multiple Passes option for more optimization. Optimizing a graphic reduces file size, but compressing it too much compromises its quality. However, optimizing curves reduces your file size and improves SWF file performance. There are third-party options available for specialized optimization of curves and points that yield different results.

There are several ways to animate content in a Flash document. Animation that uses ActionScript can produce better performance and smaller file size than animation that uses tweens at times, but sometimes not. To get the best results, try different ways of producing an effect, and test each of the options.

A higher frame rate produces smooth animation in a SWF file but it can be processor-intensive, particularly on older computers. Test your animations at different frame rates to find the lowest frame rate possible.

For information on best practices and video, see "Video conventions" on page 110. For an example of scripted animation, see the animation.fla example in the Samples/HelpExamples directory in the Flash installation folder.

Working with components in Flash Player

The new component framework lets you add functionality to components, but it can potentially add considerable file size to an application.

Components inherit from each other. One component adds size to your Flash document, but subsequent components that use the same framework do not necessarily add more size. As you add components to the Stage, the file size increases, but at some point, it levels off because components share classes and do not load new copies of those classes.

If you use multiple components that do not share the same framework, they might add substantial file size to the SWF file. For example, the XMLConnector component adds 17K to the SWF file, and TextInput components add 24K to your document. If you add the ComboBox component, it adds 28K, because it is not part of either framework. Because the XMLConnector component uses data binding, the classes add 6K to the SWF file. A document that uses all these components has 77K before you add anything else to the file. Therefore, it is a good idea to carefully consider your SWF file size when you add a new component to the document.

Components must exist in the parent SWF file's library. For example, a screen-based application must have a copy of the components it uses in its library, even if those components are required by child SWF files that are loaded at runtime. This is necessary to ensure that the components function properly, and slightly increases the download time of the parent SWF file. However, the parent library isn't inherited or shared in the SWF files that you load into the parent. Each child SWF file must download to the application with its own copy of the same components.

Using runtime shared libraries

You can improve download time by using runtime shared libraries. These libraries are usually necessary for larger applications or when numerous applications on a site use the same components or symbols. By externalizing the common assets of your SWF files, you do not download classes repeatedly. The first SWF file that uses a shared library has a longer download time, because both the SWF file and the library load. The library caches on the user's computer, and then all the subsequent SWF files use the library. This process can dramatically improve download time for larger applications.

Optimizing styles and performance

One of the most processor-intensive calls in a component framework is the `setStyle` call. The `setStyle` call executes efficiently, but the call is intensive because of the way it is implemented. The `setStyle` call is not always necessary in all applications, but if you use it, you should consider its performance impact.

To enhance performance, you can change styles before they are loaded, calculated, and applied to the objects in your SWF file. If you can change styles before the styles are loaded and calculated, you do not have to call `setStyle`.

The recommended practice to improve performance when using styles is to set properties on each object as objects are instantiated. When you dynamically attach instances to the Stage, set properties in `initObj` in the call that you make to `createClassObject()`, as the following ActionScript shows:

```
createClassObject(ComponentClass, "myInstance", 0, {styleName:"myStyle",
  color:0x99CCFF});
```

For instances that you place directly on the Stage, you can use `onClipEvent()` for each instance, or you can use subclasses (recommended). For information on subclasses, see "Creating subclasses" on page 258.

If you must restyle your components, you can improve efficiency in your application by using the Loader component. If you want to implement several styles in different components, you can place each component in its own SWF file. If you change styles on the Loader component and reload the SWF file, the components in the SWF file are recreated. When the component is recreated, the cache of styles is emptied, and the style for the component is reset and referenced again.

Note: If you want to apply a single style to all instances of a component in your SWF file, change the style globally using `_global.styles.ComponentName`.

Publishing with components

When you are planning to publish a SWF file with backward compatibility, you must have a good understanding of which components have that capability. For information about component availability in different versions of Flash Player, see the following table:

	Flash Player 6 (6.0.65.0) and earlier	Flash Player 6 (6.0.65.0)	Flash Player 7 and later
ActionScript 2.0	Supported	Supported	Supported
V2 UI component set	Not supported*	Supported	Supported
Media components	Not supported	Not supported	Supported
Data components	Not supported	Not supported	Supported

* Deselect the Optimize for Flash Player 6r65 option in Publish Settings for these components to work.

Preloading components and classes

This section describes some of the methodologies for preloading and exporting components and classes in Flash MX 2004. Preloading involves loading some of the data for a SWF file before the user starts interacting with it. Flash imports classes on the first frame of a SWF file when you use external classes, and this data is the first element to load into a SWF file. It is similar for the component classes, because the framework for components also loads into the first frame of a SWF file. When you build large applications, the loading time can be lengthy when you must import data, so you must deal with this data intelligently, as the following procedures show.

Because the classes are the first data to load, you might have problems creating a progress bar or loading animation if the classes load before the progress bar, because you probably want the progress bar to reflect the loading progress of all data (including classes). Therefore, you want to load the classes after other parts of the SWF file, but before you use components.

To select a different frame for the classes to load into a SWF file:

1. Select File > Publish Settings.
2. Select the Flash tab, and click the Settings button.
3. In the Export frame for classes text box, type the number of a new frame to determine when to load the classes.
4. Click OK.

You cannot use any classes until the playhead reaches the frame you choose to load them into. Because components require classes for their functionality, you must load components after the Export frame for ActionScript 2.0 classes. If you export for Frame 3, you cannot use anything from those classes until the playhead reaches Frame 3 and loads the data.

If you want to preload a file that uses components, you must preload the components in the SWF file. To accomplish this, you must set your components to export for a different frame in the SWF file. By default, the UI components export in Frame 1 of the SWF file.

To change the frame into which components export:

1. Select Window > Library to open the Library panel.
2. Right-click (Windows) or Control-click (Macintosh) the component in the library.
3. Select Linkage from the context menu.
4. Deselect Export in first frame.
5. Click OK.
6. Select File > Publish Settings.
7. Select the Flash tab, and click the Settings button.
8. Enter a number into the Export frame for classes text box. The classes will load into this frame.
9. Click OK.

If components do not load on the first frame, you can create a custom progress bar for the first frame of the SWF file. Do not reference any components in your ActionScript or include any components on the Stage until you load the classes for the frame you specified in Step 7.

Caution: Components must be exported after the ActionScript classes that they use.

Working with text

Computer systems have a specific code page that is regional. For example, a computer in Japan has a different code page than a computer in England. Flash Player 5 and earlier versions relied on the code page to display text; Flash Player 6 and later versions use Unicode to display text. Unicode is more reliable and standardized for displaying text because it is a universal character set that contains characters for all languages. Most current applications use Unicode.

You can use Unicode escape sequences to display special characters in Flash Player 6. However, it is possible that not all your characters display correctly if you do not load text that is UTF-8 or UTF-16 encoded (Unicode) or if you do not use a Unicode escape sequence to display the special character. For a set of Unicode code charts, see www.unicode.org/charts. For a list of commonly used escape sequences, see the table at the end of this section.

A non-Unicode application uses the operating system's code page to render characters on a page. In this case, the characters you see are specified by the code page, so the characters appear correctly only when the code page on the user's operating system matches the application's code page. This means that the code page that was used to create the SWF file needs to match the code page on the end user's computer. Using code pages is not a good idea for applications that might be used by an international audience; in this case, use Unicode instead.

Using `System.useCodepage` in your code forces the SWF file to use the system's code page instead of Unicode.

Only use this process in the following situations: when you are loading non-Unicode encoded text from an external location and when this text is encoded with the same code page as the user's computer. If both these conditions are true, the text appears without a problem. If both of these conditions are not true, use Unicode and a Unicode escape sequence to format your text. To use an escape sequence, add the following ActionScript on Frame 1 of the Timeline:

```
this.createTextField("myText_txt", 99, 10, 10, 200, 25);
myText_txt.text = "this is my text, \u00A9 2004";
```

This ActionScript creates a text field, and enters text that includes a copyright symbol (©).

You can make a SWF file use the operating system's code page, which is controlled by the `useCodepage` property. When Flash exports a SWF file, it defaults to exporting Unicode text and `System.useCodepage` is set to `false`. You might encounter problems displaying special text, or text on international systems where using the system's code page can seem to solve the problem of text incorrectly displaying. However, using `System.useCodePage` is always a last resort. Place the following line of code on Frame 1 of the Timeline:

```
System.useCodepage = true;
```

Caution: The special character displays only if the user's computer has the character included in the font that is being used. If you are not sure, embed the character or font in the SWF file.

The following table contains a number of commonly used Unicode escape sequences.

Character description	Unicode escape sequence
em-dash (—)	\u2014
registered sign (®)	\u00AE
copyright sign (©)	\u00A9
trademark sign (™)	\u2122
Euro sign (€)	\u20AC
backslash (\)	\u005C
forward slash (/)	\u002F
open curly brace ({)	\u007B
close curly brace (})	\u007D
greater than (<)	\u003C
less than (>)	\u003E
asterisk (*)	\u002A

Optimizing ActionScript in Flash Player

There are several ways that you can optimize your code for better SWF file performance, but remember that optimizing your code for Flash Player might reduce readability and consistency for code maintenance. Only practice optimize your code when necessary. Follow these guidelines to optimize your ActionScript for Flash Player:

- Avoid calling a function multiple times from a loop. It is better to include the contents of a small function inside the loop.
- Use native functions, which are faster than user-defined functions.
- Use short names for functions and variables.
- Delete your variables after you no longer use them, or set variables to `null` if you do not delete them.

  Note: Setting variables to `null` instead of deleting them can still reduce performance.

- Avoid using the `eval()` function or array access operator when possible. Often, setting the local reference once is preferable and more efficient.
- Define a `my_array.length` before a loop, rather than using `my_array.length` as a loop condition.
- Focus on optimizing loops, `setInterval`, `onEnterFrame`, and `onMouseMove`, which is where Flash Player spends a lot of time processing.

Stopping code repetition

The `onEnterFrame` event handler is useful because it can be used to repeat code at the frame rate of a SWF file. However, limit the amount of repetition that you use in a Flash file as much as possible so that you do not impact performance. For example, if you have a piece of code that repeats whenever the playhead enters a frame, it is processor-intensive. This can cause performance problems on computers that play the SWF file. This section discusses how to do this, and also how to remove movie clips and stop repeating code. If you use the `onEnterFrame` event handler for any kind of animation or repetition in your SWF files, end the `onEnterFrame` handler when you finish using it. In the following ActionScript, you stop repetition by deleting the `onEnterFrame` event handler:

```
circle_mc.onEnterFrame = function() {
  circle_mc._alpha -= 5;
  if (circle_mc._alpha<=0) {
    circle_mc.unloadMovie();
    delete this.onEnterFrame;
    trace("deleted onEnterFrame");
  }
};
```

Similarly, limit the use of `setInterval`, and remember to clear the interval when you finish using it to reduce processor requirements for the SWF file.

Guidelines for Flash applications

The best way to create different Flash applications depends on the application you create and the technology that you are using to build the application. There are guidelines that can help make the application process easier. There are also several decisions you need to make.

This section describes some guidelines and suggestions for different types of projects and applications.

For more information, see the following topics:

- "Building Flash Applications" on page 121
- "Organizing files and storing code" on page 125
- "Creating secure applications" on page 127

Building Flash Applications

An online application lets a user influence a website by interacting with it. For example, the application might collect information from the user (such as a username and password for a registration), information might be added to the site (such as in a forum), or the user might interact in real time with other site visitors (such as a chat room or interactive white board). Results from the server often appear in the SWF file, depending on the interaction. These examples are applications that involve the user and different kinds of server interaction. However, a website that does not use visitor information or data is not an application (for example, a portfolio or static informational site). Flash applications involve an interactive process between the user, a web application, and a server. The basic process is as follows:

1. A user enters information into a SWF file.
2. The information is converted into data.
3. The data is formatted and sent to a web server.
4. The information is collected by the web server and sent to an application server (for example, ColdFusion, PHP, or ASP).
5. The data is processed and sent back to the web server.
6. The web server sends the results to the SWF file.
7. The SWF file receives the formatted data.
8. Your ActionScript processes the data so the application can use it.

When you build an application, you must select a protocol for transferring data. The protocol alerts the application when data has been sent or received, in what format the data is transferred, and how it handles a server's response. After data is received in the SWF file, it must be manipulated and formatted. If you use a protocol, you do not have to worry about data being in an unexpected format. When you are transferring data using name/value pairs, you can check how the data is formatted. You need to check that the data is formatted correctly, so you do not end up receiving XML formatted data and vice versa, so the SWF file knows what data to expect and work with.

Collecting and formatting data

Applications depend on user interaction with the SWF file. Frequently, it depends on the user entering data into forms, such as using combo boxes, buttons, text fields, sliders, and so on. You might create custom input devices, use the UI Components included with Flash, or download components. You might collect data in a series of pages (sometimes each one is a screen) that are contained within the single SWF file, which the user submits to the server. Flash provides many ways you can enter and format data in Flash applications. This flexibility exists because of the capabilities you have with animation and creative control over the interface, and error checking and validation you can perform using ActionScript.

There are several benefits to using Flash to build forms to collect data:

- You have increased design control.
- You have decreased or no need for page refreshing.
- You can reuse common assets.

However, Flash might take longer to create your form than if you create it using HTML, which is a drawback.

Tip: If you want to save information that you collect from the user, you can save it in a shared object on the user's computer. Shared objects let you store data on a user's computer, which is similar to using a cookie. For more information on Shared objects, see "SharedObject class" in *Flash ActionScript Language Reference*.

Sending and processing data

The server sending data to and from Flash must be able to interpret the data that it receives. A standard format is URL-encoded data that is saved in name/value pairs, as the following example shows:

```
&name=slappy&score=398
```

A complication arises when the value being passed uses the special ampersand (&) character. In this case, the value is terminated because the data is treated as though the previous value has been terminated, as the following code shows:

```
&name=Mac&Tosh&score=136
```

If Flash loads these variables from an external site, the following variables would be defined:

```
score: 136
Tosh:
name: Mac
```

To get the code to work properly, you must encode the special characters in the URL (in this case, the ampersand [&]):

```
&name=Mac%26Tosh&score=136
/* output:
  score: 136
  name: Mac&Tosh
*/
```

You can also use `escape` and `unescape` for decoding. For a complete list of URL-encoded special characters, see www.macromedia.com/support/flash/ts/documents/url_encoding.htm.

You must typically process information before you send it to the server, so it's formatted in a way that the server understands. When the server receives the data, it can be manipulated in any number of ways and sent back to the SWF file in a format that it can accept, which can range from name/value pairs to complex objects.

Note: Your application server must have the MIME type of its output set to `application/x-www-urlform-encoded`. If that MIME type is missing, the result is usually unusable when it reaches Flash.

There are other formats for sending data, ranging from XML, Macromedia Flash Remoting, web services, server-side ActionScript (SSAS), or you can even send data using the MovieClip class's `getURL` method.

The `POST` method sends variable names and their corresponding values in the HTML header, and the `GET` method sends the name/value pairs in the browser's URL. The total number of characters you can send using the `GET` method is limited, so use the `POST` method if you are sending more than a hundred characters.

There are many ways to send data to a server and receive data using Flash. The following table shows you some of the ways:

Send data	Description
`LoadVars.send` and `LoadVars.sendAndLoad`	Sends name/value pairs to a server-side script for processing. `LoadVars.send` sends variables to a remote script and ignores any response. `LoadVar.sendAndLoad` sends name/value pairs to a server and loads or parses the response into a target `LoadVars` object.
`XML.send` and `XML.sendAndLoad`	Similar to `LoadVars`, but `XML.send` and `XML.sendAndLoad` send XML packets instead of name/value pairs.
`getURL`	Using the `getURL()` function or `MovieClip.getURL` method, it is possible to send variables from Flash to a frame or pop-up window.
Flash Remoting	Introduced in Flash MX, Flash Remoting lets you easily exchange complex information between Flash and ColdFusion, ASP.NET, Java, and more. You can also use Flash Remoting to consume web services.
Web services	Flash MX Professional includes the WebServiceConnector component that lets you connect to remote web services, send and receive data, and bind results to components. This lets Flash developers quickly create Rich Internet Applications without having to write a single line of ActionScript. It is also possible to consume remote web services using WebServiceClasses, which can require writing complex ActionScript.

There are some limitations when using web services with a SWF file, including the following:

- Flash does not support web services with more than one defined port when using the WebServiceConnector component.
- Flash does not support web services with more than one defined service when using the WebServiceConnector component.
- Flash does not support non-HTTP web services.
- Flash does not support web services on non-SOAP ports.
- Flash does not support REST web services.
- Flash does not support the `import` tag.

Note: If you are using the WebServiceConnector component in Flash MX Professional, you must place the component instance in Scene 1.

For more information about using complex data with your web service, see www.macromedia.com/support/flash/ts/documents/webserviceflaws.htm.

Adding data validation and loading

Try to validate any information you retrieve before you send that data to a server. This reduces strain on the remote server, because it does not handle as many requests when users do not fill in required fields. You should never solely rely on client-side validation in any application, so there must also be server-side validation.

Even if you build a simple registration or login form, check that the user has entered their name and password. Perform this validation before sending the request to the remote server-side script and waiting for a result. Do not rely only on server-side validation. If a user enters only a username, the server-side script has to receive the request, validate the data being sent, and return an error message to the Flash application, stating that it requires both the username and password. Likewise, if validation is performed only on the client side (within the SWF file), it might be possible for a user to hack the SWF file and manage to bypass the validation, and send data to your server in an attempt to post the bad data.

Client-side validation can be as simple as making sure that a form field has a length of at least one character, or that the user entered a numeric value and not a string. If you try to validate an e-mail address, for example, check that the text field in Flash isn't empty and contains at least the at sign (@) and dot (.) characters. For the server-side validation, add more complex validation and check that the e-mail address belongs to a valid domain.

You must create ActionScript to handle the data that loads into the SWF file from the server. After you finish loading data into a SWF file, the data can be accessed from that location. It's important to use ActionScript to check whether the data has been fully loaded. You can use callback functions to send a signal that the data has been loaded into the document.

When you load data, it can be formatted in several ways. You might load XML, and in this case, you must use the XML class methods and properties to parse the data and use it. If you use name/value pairs, the pairs turn into variables and you can manipulate them as variables.

You might receive data from a web service or from Flash Remoting. In both cases, you could receive complex data structures, such as arrays, objects, or record sets, which you must parse and bind appropriately.

Using error handling and debugging

An important part of application development is expecting and handling errors. No matter how simple your application might seem, there are always users who manage to enter data or interact with the SWF file in an unexpected way. Your application needs to be robust enough that it can anticipate certain errors and handle them accordingly.

For example, if you expect users to enter a numeric value into a text box, check that the value is numeric before you try and store or manipulate the value using code. If the value is not numeric, it is likely that the code will fail or return an unexpected result because the application cannot handle this result. Even if the user enters the data in the proper data type, you might have to validate that the data is usable before processing it. For example, if you did not check the validity of an integer before trying to perform a mathematical function, you might discover that your code fails when you try to divide a number by zero, which returns the numerical constant `Infinity`.

One of the best ways to perform error handling in Flash MX 2004 is by using the new `try-catch-finally` blocks that let you throw and catch custom errors. By creating custom error classes, you can reuse code throughout your application without having to rewrite error handling code. For more information on throwing custom errors, see "Error class" in *Flash ActionScript Language Reference*. For more information on `try-catch-finally` blocks, see `try..catch..finally` in *Flash ActionScript Language Reference*.

Organizing files and storing code

After you decide on a protocol for transferring data, you must consider how to organize your SWF files, their assets, and ActionScript. How you organize and execute your application depends greatly on its design, size, and requirements. There are guidelines to help your overall success. The following are some of the primary things you must consider before you start:

- Do you divide the SWF file into multiple SWF files, and, if so, how should they interact?
- What assets can you share across SWF files?
- What files do you dynamically load?
- How and where do you store ActionScript?

When you develop an application, try to store your server-side code and files in a logical directory structure, similar to those in an ActionScript 2.0 package. Try to arrange your code this way to keep it well organized and reduce the risk of the code being overwritten.

For larger applications, encapsulate client-server communication and services in classes. When you use classes, you benefit in the following ways:

- You can reuse the code in more than one SWF file.
- You can edit code in a central place, and update all SWF files by republishing them.
- You can create a single API that can manipulate different UI elements or other assets that perform similar functions.

Using the MVC design pattern

Many Flash developers implement the MVC (model, view, controller) design pattern when they build applications to separate the logic and data of the application from the user interface. This section defines how the design pattern works, and provides an overview of how and why you might use it for your Flash applications.

The MVC design pattern is used to separate the information, output, and data processing in the application. The application is divided into three elements: model, view, and controller; each element handles a different part of the process.

The model This part of the MVC design pattern incorporates the data and rules of the application. Much of the application's processing occurs in this part of the design pattern. The model also contains any components (such as CFCs, EJBs, and web services), and the database. Data returned is not formatted for the interface (or front end) of the application in this part of the process. This means that the returned data can be used for different interfaces (or views).

The view This particular component of the pattern handles the front end of the application (the interface with which the user interacts), and renders the model's contents. The interface specifies how the model's data is presented and outputs the view for the user to use, and lets them access or manipulate the application's data. If the model changes, the view updates to reflect those changes by either pushing or pulling data (sending or requesting data). If you create a hybrid web application (for example, one that includes Flash interacting with other applications on the page), you can consider the multiple interfaces as part of the view in the design pattern. The MVC design pattern supports handling a variety of views.

The controller The controller handles the requirements of the model and view to process and display data, and typically contains a lot of code. It calls any part of the model, depending on user requests from the interface (or view), and contains code that's specific to the application. Because this code is specific to the application, it is usually not reusable. However, the other components in the design pattern are reusable. The controller does not process or output any data, but it takes the request from the user and decides what part of the model or view components it needs to call, and determines where to send the data and what formatting is applied to the returned data. The controller ensures that views have access to parts of the model data that they must display. The controller typically transmits and responds to changes that involve the model and view.

Each part of the model is built as a self-contained component in the overall process. If you change one part of the model (for example, you might rework the interface), it reduces problems because the other parts of the process do not usually need modification. If your design pattern is created correctly, you can change the view without reworking the model or controller. If your application does not use MVC, making changes anywhere can cause a rippling effect across all your code, which requires many more changes than if you were using a specific design pattern.

There are other reasons why MVC is valuable for some applications. An important reason to use the pattern is to separate data and logic from the user interface. By separating these parts of the process, you can have several different graphical interfaces that use the same model and unformatted data. This means that you can use your application with different Flash interfaces. Perhaps the application has a Flash interface for the web, one for Pocket PC, a version for cell phones, and perhaps an HTML version that doesn't use Flash at all. Separating data from the rest of the application can greatly reduce the time it takes to develop, test, and even update more than one client interface. Similarly, adding new front ends for the same application is easier if you have an existing model to use.

Only use MVC if you build a large or complex application, such as an e-commerce website or an e-learning application. Using the architecture requires planning and understanding how Flash and this design pattern work. Carefully consider how the different pieces interact with each other; this typically involves testing and debugging. When you use MVC, testing and debugging are more involved and difficult than in typical Flash applications. If you build an application in which you need the additional complexity, consider using MVC to organize your work.

Creating secure applications

Security is an important concern that you need to address when you build applications for the Internet. There are dishonest users who might try to hack your application, whether you build a small portal site where users can log in and read articles or a large e-commerce store. For this reason, there are several steps you can consider to secure your application.

You can post data to HTTPS for data that needs to be secured. You can encrypt values in Flash before sending them to a remote server to be processed. For example, you could find a Flash library that lets you encrypt your sensitive values using an MD5 hash to help distract casual prying eyes. However, this solution means that your server-side application code must decrypt the data after it is received or compare the encrypted version of the data against an encrypted password that is stored in a database to see if they match.

Caution: Never store any information or code in a SWF file that you don't want users to see. It is easy to disassemble SWF files and view their contents using third-party software.

Perhaps the most obvious security measure for your application to add is a cross-domain policy, which prevents unauthorized domains from accessing your assets. For more information on data validation, see "Adding data validation and loading" on page 124. For more information on cross-domain policy files, see "About allowing data access between cross-domain SWF files" on page 289 and "About allowing cross-domain data loading" on page 290.

Projects and version control guidelines

Projects in Flash introduce a way for members on a team to work together on a single Flash application or project. A project file remembers each of the files it contains, and lets you incorporate some SourceSafe capabilities into your applications, which helps you keep backups of modified files.

Note: Flash MX Professional 2004 does not support SourceSafe for version control on the Macintosh.

Using projects

You can group multiple files into a single project file using the Project panel in Flash MX Professional 2004. This helps simplify application building, where managing related files could get complex and confusing. You can define a site for your work, create a Flash Project file (FLP), and then upload everything to the server so that a team can work on the project.

Version control features help you ensure that you use the correct current files when authoring, and that certain files are not overwritten. When multiple authors work on the same project, you can check that only one person has the file checked out and, during that time, another person cannot overwrite the file.

You can likely use your current source control software with Flash, but you might not be able to integrate it with the Project panel. Microsoft Visual SourceSafe is currently supported. Other software programs can manage and control your Flash documents, but you probably cannot integrate them with the Project panel. For more information on projects in Flash, see "Creating and managing projects (Flash Professional only)" in *Using Flash*.

Using version control

Version control lets you check files in and out of your repository, and check that only one person is working on a file at a certain time. Other benefits include the ability to revert to older versions of the files, so if your FLA file becomes corrupted or spontaneously stops working, you can revert to an older (working) version.

There are certain ways that you can organize your project's workflow. This section describes the best practices to follow when working with Flash projects and version control. For more information on using version control in Flash, see "Using version control with projects (Flash Professional only)" in *Using Flash*.

Administrating projects

Assign an administrator to the project. This individual is responsible for creating and maintaining the project's structure. For example, documents are split up logically using folders to combine similar files. Typically, several authors work on one Flash project. The administrator confirms changes that are made to the project's structure, which encourages project stability.

Caution: The administrator is the only person who changes the project file and structure.

The project's administrator defines the site, and creates the Flash Project (FLP), main FLA document, and any subdirectories for the project's assets. These directories might include media, images, or classes that dynamically load into the project. The administrator uploads everything to the server. The administrator also creates a clear structure for the project, and communicates how it works and where to add additional assets (such as class and image files) to everyone who is working on the application.

Authoring projects

Authors on a Flash project do not change the project root, directory structure of the project, or the site. This includes adding, removing, or changing subdirectory names, or adding additional subdirectories to the project on their local computer. If individual authors change the site or project structure, the local files are out of sync with those on the server. This causes problems in the application, such as class path and missing file errors, and so on. Individual authors can copy assets to the subdirectory files that the project's administrator creates.

Each author on a Flash project selects File > Open from Site, selects the name of the site, and then selects the project's FLP file. Then the author updates the project with any missing files. This ensures that the author is working with the latest version of the site. When the author selects Yes, all the project files download to the author's local computer, so the structure on the local computer matches the structure on the server.

Changing structure

When the project's structure needs to be changed, authors check in all their files. The project's administrator checks out all the files to make any necessary changes. After this is done, each person working on the project deletes the root folder of their own local copy of the project. Each individual author should use File > Open from Site to download a new copy of the site. This helps reduce errors when working with the project from accidentally using legacy files, and reduces similar versioning problems.

Guidelines for accessibility in Flash

Screen readers are complex, and you can easily encounter unexpected results in FLA files developed for use with screen readers, which is software that visually impaired users run to read websites aloud. Flash applications must be viewed in Internet Explorer on Windows, because Microsoft Active Accessibility (MSAA) support is limited to this browser.

For more information, see the following topics:

- "Creating accessible sites" on page 130
- "Using screen readers" on page 131
- "Exposing SWF file structure and navigation" on page 131
- "Controlling descriptions and repetition" on page 132
- "Using color" on page 132
- "Ordering, tabbing, and the keyboard" on page 133
- "Handling audio and animation" on page 134
- "Extending Flash and accessibility" on page 134
- "Working with accessibility and components" on page 134
- "Testing frequently and making changes" on page 135

Creating accessible sites

Flash Player uses Microsoft Active Accessibility (MSAA) to expose Flash content to screen readers. MSAA is a Windows-based technology that provides a standardized platform for information exchange between assistive technologies, such as screen readers, and other applications. Events (such as a change in the application) and objects are visible to screen readers by using MSAA.

Making a website accessible involves several different criteria:

Expose the information to screen readers Use the techniques outlined in this section to expose parts of your SWF files to screen readers.

Make text or images realizable Some visitors might have difficulty reading small text or seeing small graphics. Allow users to zoom in on these elements, taking advantage of scalable vector graphics in SWF files.

Provide audio narration Consider providing an audio narration for visitors without a screen reader, or where screen readers might not work, such as with video content.

Provide captions for audio narrations Some visitors might not be able to hear an audio narration for your site or a video. Consider providing captions for these visitors.

Do not rely on color to communicate information Many visitors might be color blind. If you rely on color to communicate information (such as: Click the green button to go to page 1, click the red button to go to page 2), provide text or speech equivalents.

Section 508 is legislation in the United States that provides guidelines for making information accessible to people with disabilities, such as vision impairments. Section 508 specifically addresses the need for websites to be accessible in several ways. Some websites, including all federal websites, must comply with these guidelines. If a SWF file does not communicate all of the information to the screen reader, the SWF file is no longer Section 508-compliant. For information on Section 508, see www.section508.gov.

Historically, many online presentations (such as videos) provide alternate ways for visually impaired visitors to access the content. An example of this would be a textual description of a video. However, Flash provides textual information directly to the screen reader. Although this usually means you need to make additional settings or ActionScript in a FLA file, you do not have to create a completely separate version.

Parts of your SWF file can be exposed to screen readers. Text elements (such as text fields, static text, and dynamic text), buttons, movie clips, components, and the entire SWF file can be interpreted by MSAA -compliant screen readers. The following sections discuss how to work with Flash and screen readers.

For more information on Accessibility and web standards, see: www.w3.org/WAI/. These standards and guidelines do not offer much assistance when working with Flash content, but describe what factors you must address when you create accessible HTML websites, and some of this information applies to Flash.

Using screen readers

A screen reader is software that lets your visitors hear a description of the contents of web pages. Text is read aloud using specially designed software. Obviously, a screen reader can only interpret textual content. However, any descriptions that you provide for the overall SWF file, movie clips, images, or other graphical content are also read aloud. It is important that you write descriptions for the important images and animations so that the screen reader can also interpret these assets in your SWF file. This is the SWF file equivalent to *alt* text in an HTML web page.

Freedom Scientific JAWS for Windows is one of the most common screen readers available; version 4.5 and later is compatible with Flash Player 6 (6.0.21.0) and later. Window Eyes by GW Micro is another one of the most commonly used screen readers, and version 4.2 and later is supported by Flash Player 6 (6.0.21.0) and later. Accessible content is interpreted differently by each of these screen readers, and might also behave differently on different players. You can download free but time-limited demo software from the following websites.

- Window Eyes from www.gwmicro.com
- JAWS for Windows from www.freedomscientific.com

You can also try using Connect Outloud from Freedom Scientific, which is software based on JAWS, but it is designed only for web content. For more information, see www.freedomscientific.com/fs_products/software_connect.asp. To use Connect Outloud, you must download a DLL for this software to work with Flash content from the following location: www.freedomscientific.com/fs_products/software_connectinter.asp.

Note: Flash Player 7 does not work with all screen reader technologies. It is up to the third-party software provider to handle the information provided by MSAA.

Exposing SWF file structure and navigation

Because of the highly visual nature of some SWF files, the layout and navigation of the page can be complex and difficult for screen readers to translate. An overall description of the SWF file is important to communicate information about its structure and how to navigate through the site's structure. You can provide this description by clicking the Stage and entering a description into the Accessibility panel. You could also create a separate area of the site to provide this description or overview.

Note: If you enter a description for the main SWF file, this description is read each time the SWF file refreshes. You can avoid this redundancy by creating a separate informational page.

Inform the user about any navigational elements that change in the SWF file. Perhaps an extra button is added, or the text on the face of a button changes, and this change is read aloud by the screen reader. Flash Player 7 supports updating these properties using ActionScript. This means that you can update the accessibility information in your applications if the content changes at runtime. For more information on updating accessible properties at runtime, see "Creating accessibility with ActionScript" in *Using Flash*.

Controlling descriptions and repetition

Designers and developers can assign descriptions for the animations, images, and graphics in a SWF file. Provide names for graphics so the screen reader can interpret them. If a graphic or animation does not communicate vital information to the SWF file (perhaps it is decorative or repetitive), or you outlined the element in the overall SWF file description, do not provide a separate description for that element. Providing unnecessary descriptions can be confusing to users who use screen readers. For more information on assigning names and descriptions, see "Using Flash to enter accessibility information for screen readers" in *Using Flash*.

Note: If you divide text or use images for text in your SWF files, provide either a name or description for these elements.

If you have several nested movie clips that serve a single purpose or convey one idea, ensure that you do the following:

- Group these elements in your SWF file.
- Provide a description for the parent movie clip.
- Make all the child movie clips inaccessible.

This is extremely important, or the screen reader tries to describe all the irrelevant nested movie clips, which will confuse the user, and might cause the user to leave your website. Make this decision whenever you have more than one object, such as many movie clips, in a SWF file. If the overall message is best conveyed using a single description, provide a description on one of the objects, and make all the other objects inaccessible to the screen reader.

Looping SWF files and applications cause screen readers to constantly refresh. This occurs because the screen reader is detecting that there is new content on the page, and because it thinks the content is updated, it returns to the top of the web page and starts rereading the content. You should make any looping or refreshing objects that do not have to be reread inaccessible to screen readers.

Note: Do not type a description in the Description field of the Accessibility panel for instances (such as text) that the screen reader reads aloud.

Using color

It is tempting to use a wide array of colors in a Flash SWF file. It is possible to use all these colors in an accessible SWF file, but you must make some decisions about it. For example, you must not only rely on color to communicate particular information or directives to users. A color-blind user cannot operate a page if it asks to click on the blue area to launch a new page or the red area to hear music. Offer text equivalents on the page or in an alternate version to make your site accessible. Also, check that there is significant contrast between foreground and background colors to assist users who have difficulty seeing particular colors, to enhance readability. If you place light gray text on a white background, users cannot easily read it. Similarly, small text, which is commonly found in many Flash websites, is difficult for many visitors to read. Using high-contrast and large or resizable text benefits most users, even those without impairments.

Ordering, tabbing, and the keyboard

The reading order and tabbing are possibly the two most important considerations for making accessible Flash websites. When you design an interface, the order that it appears on the page might not match the order in which the screen reader describes each instance. There are ways you can control and test reading order, as well as control tabbing in the SWF file.

Controlling reading order

Screen readers sometimes read assets in an unpredictable order. The default reading order does not always match the placement of your assets or the visual layout of the page. If you keep the layout simple, this can help create a logical reading order without the need for ActionScript. However, this isn't always possible and doesn't necessarily work as expected. You have more control over the order in which your content is read if you use ActionScript. For more information on controlling reading order using ActionScript, see "Creating accessibility with ActionScript" in *Using Flash*.

Because the default reading order is not predictable, use ActionScript and test the reading order in your SWF files.

Caution: Do not miss ordering a single instance in your SWF file, or the reading order reverts to the default (and unpredictable) reading order.

Controlling tabbing and content

Visitors who rely on screen readers to describe a site's content typically use tabbing and keyboard controls to navigate around the operating system and web pages, because using the mouse is not useful when the screen cannot be seen. You should offer intelligent tabbing control in accessible SWF files using the `tabIndex` and `tabEnabled` properties with the movie clip, button, text field, or component instances. In addition to tabbing, you can use any key press actions to navigate through the SWF file, but you must communicate that information using the Accessibility panel. Use the `Key` class in ActionScript to add keypress scripts to the SWF file. Select the object for which you want to use the keypress script, and add the shortcut key in the Shortcut field on the Accessibility panel. Add keyboard shortcuts to essential and frequently used buttons in your SWF file.

Note: Avoid invisible buttons in accessible SWF files, because screen readers do not recognize these buttons. (Invisible buttons are buttons for which you define only a hit area, the clickable region, for the button.)

Similarly, give your users control over the content of the SWF file. Many SWF files have a rapid succession of information, and screen readers frequently cannot keep up with this pace. It is simple to resolve this problem by handing control of the process to your user. Provide controls for the SWF file, letting the user step through the file at their own pace using buttons, and letting them pause the process if necessary. Users with screen readers might interpret the content at a slower pace than other users, so giving control to the user is important in these cases.

Handling audio and animation

Many SWF files contain audio, video, and narrations, because it is easy and robust to deliver this media using Flash. When you provide audio narrations or video that contains speech, it is important to provide captions for those users who cannot hear. You can use text fields in Flash, import video that contains captions, or even use an XML caption file. For information on using captions in Flash, see "Accessibility for hearing-impaired users" in *Using Flash*. For information on using Hi-Caption SE and the Hi-Caption Viewer component, see www.macromedia.com/software/flash/extensions. This third-party extension lets you create captions that you save in an XML file and load into the SWF file at runtime, among other advanced controls.

Extending Flash and accessibility

With the new extensibility layer in Flash MX 2004, developers can create extensions that enable advanced authoring with little effort. This lets third-party companies develop extensions that involve accessibility. You have several options for validating your SWF files or adding captions.

For example, a validation tool can look through your SWF file for missing descriptions. It checks to see if a description is added for a group of instances, or if text has a label for the instance, and tells you about any problems. The tool also examines the reading order in your SWF file, and finds all instances that must be specified. Reading order can be specified after the SWF file is analyzed using a dialog box.

For information on the currently available third-party extensions, see www.macromedia.com/software/flash/extensions.

Working with accessibility and components

Window Eyes 4.2 or later, and JAWS 4.5 or later support the following components:

- Alert
- Button
- CheckBox
- ComboBox
- DataGrid
- Label
- ListBox
- RadioButton
- TextArea
- TextInput
- Window

Note: Screen readers do not recognize Window and Alert instances if you use them as pop-up windows.

Thoroughly test component instances in your applications with a screen reader, because some components have problems when you use them with screen readers. In particular, test how the screen reader reads components aloud when the component is in an opened and closed state. Notice how the reader announces components when using key press actions to open, close, and select items in the component.

Testing frequently and making changes

It is strongly advised that you test any SWF file that is intended for use with screen readers. Test your SWF files when each new version of Flash Player is released, including minor revisions, such as Flash Player 7 (7.0.19.0). In addition to testing the SWF file with different Flash Players, also test the SWF file with both Window Eyes and JAWS for Windows screen readers. These two screen readers handle SWF files differently, so you could get different results for the user experience.

Note: You do not have to test different browsers, because the technology used to expose SWF files to screen readers (MSAA) is supported only by Internet Explorer on Windows.

To test your sites with a screen reader:

1. Open your site in a browser without a screen reader, and navigate through your site without using the mouse.
2. Open your site in Window Eyes, and navigate through your site, listening to how content is read in the screen reader.
3. Try navigating your website after turning off your monitor and using only the screen reader.
4. Repeat step 2 and step 3 using JAWS for Windows.
5. If you use audio narration, test your site without speakers.
6. Test your site with several target visitors.

When listening to your SWF file using a screen reader, check the following points:

- Is the reading order correct?
- Do you have descriptions for shortcuts in your SWF file?
- Do you have adequate and complete descriptions for the elements in the interface?
- Do you have adequate descriptions for navigating the site's structure?
- Is the SWF file content read when it is updated or refreshed?
- If you change the context of any elements on the Stage (such as a button that changes from Play to Pause), is that change announced by the screen reader?

There is no official tool available for validating SWF files, unlike HTML validation. However, some third-party tools exist to help you validate the file. For more information on these extensions, see www.macromedia.com/software/flash/extensions.

Advertising with Flash

Many opportunities exist for creating interactive and engaging advertisements using SWF files. Macromedia recommends that you follow several guidelines when you produce Flash advertisements, based on standards set up by the Interactive Advertising Bureau (IAB).

Using recommended dimensions

It is recommended that you use the Interactive Advertising Bureau (IAB) guidelines to set dimensions for your Flash advertisements. The following table lists the recommended Interactive Marketing Unit (IMU) ad formats measurements:

Type of advertisement	Dimensions (pixels)
Wide skyscraper	160 x 600
Skyscraper	120 x 600
Half-page ad	300 x 600
Full banner	468 x 60
Half banner	234 x 60
Micro bar	88 x 31
Button 1	120 x 90
Button 2	120 x 60
Vertical banner	120 x 240
Square button	125 x 125
Leaderboard	728 x 90
Medium rectangle	300 x 250
Square pop-up	250 x 250
Vertical rectangle	240 x 400
Large rectangle	336 x 280
Rectangle	180 x 150

When you create a new FLA file from a template (Select File > New, and click the Templates tab), you see many of these sizes.

Creating SWF file advertisements

Optimize your graphics as much as possible. For more information on optimizing graphics and animations, see "Optimizing graphics and animation" on page 114. Make your SWF file banner advertisements 15K or smaller. Create a GIF banner advertisement in Flash that is 12K or smaller. Limit looping banner advertisements to three repetitions. Many websites adopt the standardized file size recommendations as advertising specifications.

Use the `GET` command to pass data between an advertisement and a server, and do not use the `POST` command. For more information on GET and POST, see `getURL()` in *Flash ActionScript Language Reference*.

Note: Remember to provide control to the user. If you add sound to an advertisement, also add a mute button. If you create a transparent Flash ad that hovers over a web page, always provide a button to close the advertisement for its entire duration.

Tracking advertisements

Several leading advertising networks now support standardized tracking methods in Flash SWF files. The following guidelines describe the supported tracking methodology:

Create a button or movie clip button You should use standardized dimensions outlined by the IAB. For a list of standardized dimensions, see www.iab.net/standards. For more information on creating a button in Flash, see "Creating buttons" in *Using Flash*.

Add a script to the button This script executes when a user clicks the banner. You might use the `getURL()` function to open a new browser window. For example, you might add the following to Frame 1 of the Timeline:

```
this.myButton_btn.onRelease = function(){
  getURL(clickTAG, "_blank");
};
```

You might add the following code to Frame 1 of the Timeline:

```
this.myButton_btn.onRelease = function() {
  if (clickTAG.substr(0, 5) == "http:") {
    getURL(clickTAG);
  }
};
```

The `getURL()` function adds the variable passed in the `object` and `embed` tags, and then sends the browser that is launched to the specified location. The server hosting the ad can track clicks on the advertisement. For more information on using the `getURL()` function, see `getURL()`in *Flash ActionScript Language Reference*.

Assign clickTAG code for tracking This code tracks the advertisement and helps the network serving the ad to track where the ad appears and when it is clicked.

The process is the standard way of creating an advertising campaign for a typical Flash advertisement. If you assign the `getURL()` function to the banner, you can use the following process to add tracking to the banner. The following example lets you append a variable to a URL string to pass data, which lets you set dynamic variables for each banner, instead of creating a separate banner for each domain. This means that you can use a single banner for the entire campaign, and any server that is hosting the ad can track the clicks on the banner.

In the `object` and `embed` tags in your HTML, you would add code similar to the following example:

```
<EMBED src="your_ad.swf?clickTAG=
  http://adnetwork.com/tracking?http://www.destinationURL.com">
```

And you would add the following code in your HTML:

```
<PARAM NAME=movie VALUE="your_ad.swf?clickTAG
  =http://adnetwork.com/tracking?http://www.destinationURL.com">
```

For more information on advanced tracking techniques, see the Rich Media Advertising Center at www.macromedia.com/devnet/rich_media_ads.

To download the Rich Media Tracking Kit, which includes examples and documentation, see www.macromedia.com/resources/richmedia/tracking.

To learn more about and download the Flash Ad Kit, which helps you deliver integrated and sophisticated advertisements, see www.macromedia.com/devnet/rich_media_ads/articles/flash_ad_kit.html.

Testing your ads

Ensure that your SWF file ad works on the most common browsers, and, in particular, the ones that your target audience use. Some users might not have Flash Player installed or they might have JavaScript disabled. Plan for these circumstances by having a replacement (default) GIF image or other scenarios for these users. For more information on detecting Flash Player, see "Setting publish options for the Flash SWF file format" in *Using Flash*. Be sure to give the user control of the SWF file. Let the user control any audio in the ad. If the advertisement is a borderless SWF file that hovers over a web page, let the user close the advertisement immediately and for the duration of the ad.

For the latest information on Flash Player version penetration for different regions, go to www.macromedia.com/software/player_census/flashplayer/version_penetration.html.

CHAPTER 4
Writing and Debugging Scripts

Adding scripts to your Flash applications enables rich functionality. In Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004, you have two choices:

- You can write scripts that are embedded in your Flash document (FLA file). To write embedded scripts, you use the Actions panel and attach your scripts to a button, movie clip, or frame in the Timeline (see "Controlling when ActionScript runs" on page 140).
- You can write scripts that are stored externally on your computer. External scripts are run when the scripts are called in your FLA file. To write external script files, you can use any text editor or code editor.

  In Flash Professional, you can also use the Script window. For more information, see "Using the Actions panel and Script window" on page 140.

When you use the Actions panel or the Script window to write scripts, the ActionScript editor lets you check syntax for errors, automatically format code, and use code hints to help you complete syntax. In addition, the punctuation balance feature helps you pair parentheses [()], braces ({}), or brackets ([]). For more information, see "Using the ActionScript editor" on page 144.

As you work on a document, test it often to ensure that it plays as smoothly as possible and as expected. You can use the Bandwidth Profiler to simulate how your document will appear at different connection speeds (see "Testing document download performance" in Using Flash Help). To test your scripts, you use a special debugging version of Flash Player that helps troubleshooting. If you use good authoring techniques in your ActionScript, your scripts will be easier to troubleshoot when something behaves unexpectedly. For more information, see "Debugging your scripts" on page 153.

If you are writing ActionScript 2.0 class files, see Chapter 10, "Creating Custom Classes with ActionScript 2.0," on page 247.

Controlling when ActionScript runs

When you write a script, you use the Actions panel to attach the script to a frame on a Timeline or to a button or movie clip on the Stage. Scripts attached to a frame run, or *execute*, when the playhead enters that frame. However, scripts attached to the first frame of a SWF file can behave differently from those attached to subsequent frames because the first frame in a SWF file is rendered incrementally—objects are drawn on the Stage as they download into Flash Player—and this can affect when scripts execute. All frames after the first frame are rendered at the same time, when every object in the frame is available.

Scripts attached to movie clips or buttons execute when an event occurs. An *event* is an occurrence in the SWF file such as a mouse movement, a keypress, or a movie clip being loaded. You can use ActionScript to find out when these events occur and execute specific scripts, depending on the event. For more information, see Chapter 5, "Handling Events," on page 167.

Using the Actions panel and Script window

You can embed Flash scripts in your FLA file or store them as external files. It's a good idea to store as much of your ActionScript code in external files as possible, which makes it easier to reuse code in multiple FLA files. In your FLA file, you can create a script that uses `#include` statements to access the code you've stored externally. Use the .as suffix to identify your scripts as ActionScript (AS) files.

You can use ActionScript 2.0 to create custom classes. You must store custom classes in external AS files and use `import` statements in a script to get the classes exported into the SWF file, instead of using `#include` statements. For more information on ActionScript 2.0 and importing classes, see "New object-oriented programming model" on page 21 and "Importing classes" on page 271.

Note: ActionScript code in external files is compiled into a SWF file when you publish, export, test, or debug a FLA file. Therefore, if you make any changes to an external file, you must save the file and recompile any FLA files that use it.

When you embed ActionScript code in your FLA file, you can attach code to frames and to object instances (such as movie clips). One method is to attach embedded ActionScript to the first frame of the Timeline whenever possible so you don't have to search through a FLA file to find all your code; it is centralized in one location. Another method is to create a layer called Actions and place your code there. If you place code on other frames or attach it to objects, you can find your code on that layer.

To create scripts that are part of your document, you enter ActionScript directly into the Actions panel. To create external scripts, use your preferred text editor or, in Flash Professional, you can use the Script window (File > New > ActionScript File). When you use the Actions panel or Script window, you are using the same ActionScript editor and are typing your code in the Script pane at the right side of the panel or window. Instead of typing code into the Actions panel, you can also select or drag actions from the Actions toolbox to the Script pane.

To display the Actions panel:

Do one of the following:

- Select Window > Development Panels > Actions.
- Press F9.

(Flash Professional only) To display the Script window:

Do one of the following:

- To begin writing a new script, select File > New > ActionScript File.
- To open an existing script, select File > Open, and then open an existing AS file.
- To edit a script that is already open, click the document tab that shows the script's name. (Document tabs are supported only in Microsoft Windows.)

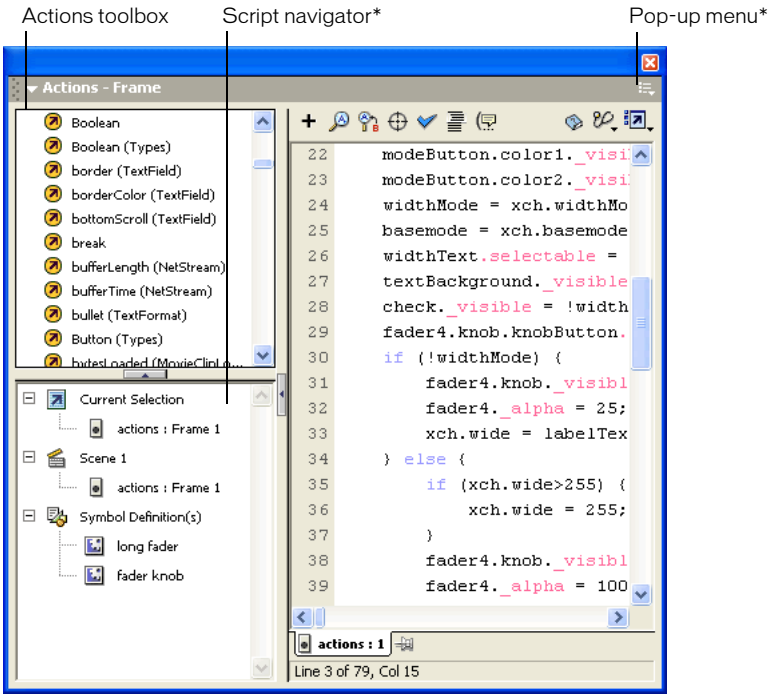
For more information, see the following topics:

- "About the ActionScript editor environment" on page 141
- "Managing scripts in a FLA file" on page 143

About the ActionScript editor environment

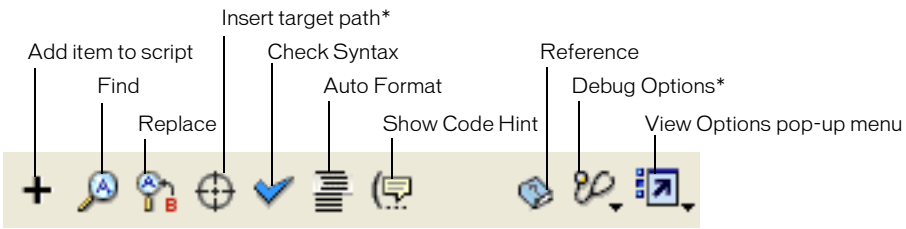
The ActionScript editor environment consists of two sections. The section on the right is the Script pane; this is the area where you type your code. The section on the left is an Actions toolbox that contains an entry for each ActionScript language element.

In the Actions panel, the Actions toolbox contains a Script navigator, which is a visual representation of the locations in the FLA file that have associated ActionScript; you can navigate through your FLA file to locate ActionScript code. If you click an item in the Script navigator, the script associated with that item appears in the Script pane, and the playhead moves to that position on the Timeline. If you double-click an item in the Script navigator, the script gets *pinned* (locked in place). For more information, see "Managing scripts in a FLA file" on page 143.



* Actions panel only

There are also several buttons above the Script pane:

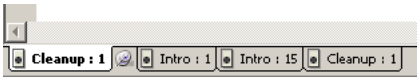


* Actions panel only

You edit actions, enter parameters for actions, or delete actions directly in the Script pane. You can also double-click on an item in the Actions toolbox or the Add (+) button above the Script pane to add actions to the Script pane. After the ActionScript is added to the pane, some actions (where applicable) display code hints in the Script pane to help you complete the syntax. For more information on code hints, see "Using code hints" on page 147.

Managing scripts in a FLA file

If you don't centralize your code within a FLA file in one location, you can pin multiple scripts in the Actions panel to make it easier to move among them. In the following figure, the script associated with the current location on the Timeline is on Frame 1 of the layer named Cleanup. (The tab at the far left always follows your location along the Timeline.) That script is also pinned (it is shown as the rightmost tab). Two other scripts are pinned; one on Frame 1 and the other on Frame 15 of the layer named Intro. You can move among the pinned scripts by clicking on the tabs or by using keyboard shortcuts, such as Control+Shift+. (period). Moving among pinned scripts does not change your current position on the Timeline.



Tip: If the content in the Script pane doesn't change to reflect the location that you select on the Timeline, the Script pane is probably showing a pinned script. Click the left tab at the lower left of the Script pane to show the ActionScript associated with your location along the Timeline.

To pin a script:

1. Position your pointer on the Timeline so the script appears in a tab at the lower left of the Script pane in the Actions panel.
2. Do one of the following:
   - Click the pushpin icon to the right of the tab.
   - Right-click (Windows) or Control-click (Macintosh) on the tab, and select Pin Script.
   - Select Pin Script from the Options pop-up menu (at the upper right of the panel).
   - With the pointer focused in the Script pane, press Control+= (equal sign) in Windows or Command+= on the Macintosh.

To unpin one or more scripts:

Do one of the following:

- If a pinned script appears in a tab at the lower left of the Script pane in the Actions panel, click the pushpin icon on the right of the tab.
- Right-click (Windows) or Control-click (Macintosh) on a tab, and select Close Script or Close All Scripts.
- Select Close Script or Close All Scripts from the Options pop-up menu (at the upper right of the panel).
- With the pointer focused in the Script pane, press Control+-(minus sign) in Windows or Command+- on Macintosh.

To use keyboard shortcuts with pinned scripts:

- You can use the following keyboard shortcuts to work with pinned scripts:

Action	Windows shortcut key	Macintosh shortcut key
Pin script	Control+= (equal sign)	Command+=
Unpin script	Control+- (minus sign)	Command+-
Move focus to tab on the right	Control+Shift+. (period)	Command+Shift+.
Move focus to tab on the left	Control+Shift+, (comma)	Command+Shift+,
Unpin all scripts	Control+Shift+- (minus)	Command+Shift+-

Using the ActionScript editor

When you use the ActionScript editor (in either the Actions Panel or, in Flash MX Professional 2004, the Script window), you can use several features to help you write syntactically correct code and set preferences for code formatting and other options. The following capabilities are discussed in this section:

- "Syntax highlighting" on page 144
- "Writing code that triggers code hints" on page 145
- "Using code hints" on page 147
- "Using Escape shortcut keys" on page 149
- "Checking syntax and punctuation" on page 150
- "Formatting code" on page 151

Syntax highlighting

In ActionScript, as in any language, *syntax* is the way elements are put together to create meaning. If you use incorrect ActionScript syntax, your scripts cannot work.

When you write scripts in Flash MX 2004 and Flash MX Professional 2004, commands that are not supported by the version of the player you are targeting appear in yellow in the Actions toolbox. For example, if the Flash Player SWF version is set to Flash 6, ActionScript that is supported only by Flash Player 7 appears in yellow in the Actions toolbox. (For information on setting the Flash Player SWF version, see "Setting publish options for the Flash SWF file format" in *Using Flash*.)

You can also set a preference to have Flash "color-code" parts of your scripts as you write them, which brings attention to typing errors. For example, suppose you set the Syntax coloring preference to have keywords appear in deep blue. While you type code, if you type `var`, the word `var` appears in blue. However, if you mistakenly type `vae`, the word `vae` remains black, which shows that you made a typing error. For information on keywords, see "Keywords and reserved words" on page 32.

To set preferences for syntax coloring as you type, do one of the following:

- Select Edit > Preferences, and specify Syntax coloring settings on the ActionScript tab.
- In the Actions panel, select Preferences from the Options pop-up menu (at the upper right of the panel) and specify Syntax coloring settings on the ActionScript tab.
- With the pointer focused in the Script pane, press Control-U (Windows) or Command-U (Macintosh).

You can change the color settings for keywords, comments, identifiers and strings. For information on identifiers and strings, see "Terminology" on page 24 and "String data type" on page 35. For information on commenting, see "Comments" on page 31.

Writing code that triggers code hints

When you work in the ActionScript editor (either in the Actions panel or Script window), Flash can detect what action you are entering and display a *code hint*—a tooltip that contains the complete syntax for that action or a pop-up menu that lists possible method or property names. Code hints appear for parameters, properties, and events when you strictly type or name your objects, as discussed in the rest of this section. Code hints also appear if you double-click an item in the Actions toolbox or the Add (+) button above the Script pane to add actions to the Script pane. For information on using code hints when they appear, see "Using code hints" on page 147.

Note: Code hinting is enabled automatically for native classes that don't require you to create and name an instance of the class, such as Math, Key, Mouse, and so on.

Strictly typing objects to trigger code hints

When you use ActionScript 2.0, you can strictly type a variable that is based on a built-in class, such as Button, Array, and so on. If you do so, the ActionScript editor displays code hints for the variable. For example, suppose you type the following code:

```
var names:Array = new Array();
names.
```

As soon as you type the period (.), Flash displays a list of methods and properties available for Array objects because you have typed the variable as an array. For more information on data typing, see "Strict data typing" on page 41. For information on using code hints when they appear, see "Using code hints" on page 147.

Using suffixes to trigger code hints

If you use ActionScript 1, or you want to display code hints for objects you create without strictly typing them(see "Strictly typing objects to trigger code hints" on page 145), you must add a special suffix to the name of each object when you create it. For example, the suffixes that trigger code hinting for the Array class and the Camera class are `_array` and `_cam`, respectively. For example, if you type the following code:

```
var my_array = new Array();
var my_cam = Camera.get();
```

Then you can type either of the following (the variable name followed by a period):

```
my_array.
my_cam.
```

You will see code hints for the Array and Camera object, respectively, appear.

For objects that appear on the Stage, use the suffix in the Instance Name text box in the Property inspector. For example, to display code hints for MovieClip objects, use the Property inspector to assign instance names with the suffix `_mc` to all MovieClip objects. Then, whenever you type the instance name followed by a period, code hints appear.

Although suffixes are not required for triggering code hints when you strictly type an object, using them consistently helps make your code understandable.

The following table lists the suffixes required for support of automatic code hinting:

Object type	Variable suffix
Array	_array
Button	_btn
Camera	_cam
Color	_color
ContextMenu	_cm
ContextMenuItem	_cmi
Date	_date
Error	_err
LoadVars	_lv
LocalConnection	_lc
Microphone	_mic
MovieClip	_mc
MovieClipLoader	_mcl
PrintJob	_pj
NetConnection	_nc
NetStream	_ns
SharedObject	_so
Sound	_sound
String	_str
TextField	_txt
TextFormat	_fmt
Video	_video
XML	_xml

Object type	Variable suffix
XMLNode	_xmlnode
XMLSocket	_xmlsocket

For information on using code hints when they appear, see "Using code hints" on page 147.

Using comments to trigger code hints

You can also use ActionScript comments to specify an object's class for code hinting. The following example tells ActionScript that the class of the instance `theObject` is Object, and so on. If you were to enter `mc` followed by a period after these comments, a code hint would display the list of MovieClip methods and properties; if you were to enter `theArray` followed by a period, a code hint would display a list of Array methods and properties; and so on.

```
// Object theObject;
// Array theArray;
// MovieClip mc;
```

However, Macromedia recommends that, instead of this technique, you use strict data typing (see "Strictly typing objects to trigger code hints" on page 145) or suffixes (see "Using suffixes to trigger code hints" on page 145) because these techniques enable code hinting automatically and make your code more understandable.

Using code hints

Code hints are enabled by default. By setting preferences, you can disable code hints or determine how quickly they appear. When code hints are disabled in preferences, you can still display a code hint for a specific command.

To specify settings for automatic code hints:

Do one of the following:

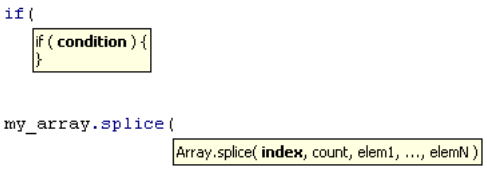
- Select Edit > Preferences, and then enable or disable Code Hints on the ActionScript tab.
- In the Actions panel, select Preferences from the Options pop-up menu (at the upper right of the panel) and enable or disable Code hints on the ActionScript tab.

If you enable code hints, you can also specify a delay in seconds before the code hints should appear. For example, if you are new to ActionScript, you might prefer no delay so that code hints always appear immediately. However, if you usually know what you want to type and need hints only when you use unfamiliar language elements, you can specify a delay so that code hints don't appear when you don't plan to use them.

To work with tooltip-style code hints:

1. Display the code hint by typing an opening parens [(] after an element that requires parentheses (such as a method name, a command such as `if` or `do while`, and so on).

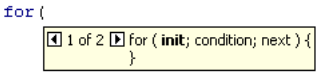
   The code hint appears.



   Note: If a code hint doesn't appear, make sure you haven't disabled code hints on the ActionScript tab. If you want to display code hints for a variable or object you created, make sure that you have named your variable or object correctly (see "Using suffixes to trigger code hints" on page 145) or that you have strictly typed your variable or object (see "Strictly typing objects to trigger code hints" on page 145).

2. Enter a value for the parameter. If there is more than one parameter, separate the values with commas or, for functions or statements such as the `for` loop, semicolons.

   Overloaded commands (functions or methods that can be invoked with different sets of parameters) such as `gotoAndPlay()` or `for` display an indicator that lets you select the parameter you want to set. Click the small arrow buttons or press Control+Left Arrow and Control+Right Arrow to select the parameter.

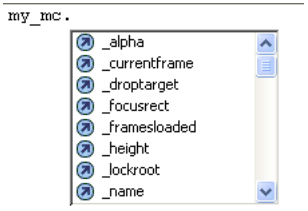


3. To dismiss the code hint, do one of the following:
   - Type a closing parens [)].
   - Click outside the statement.
   - Press Escape.

To work with menu-style code hints:

1. Display the code hint by typing a period after the variable or object name.

   The code hint menu appears.



Note: If a code hint doesn't appear, make sure you haven't disabled code hints on the ActionScript tab. If you want to display code hints for a variable or object you created, make sure that you have named your variable or object correctly (see "Using suffixes to trigger code hints" on page 145) or that you have strictly typed your variable or object (see "Strictly typing objects to trigger code hints" on page 145).

2. To navigate through the code hints, use the Up and Down Arrow keys.
3. To select an item in the menu, press Enter or Tab, or double-click the item.
4. To dismiss the code hint, do one of the following:
   - Select one of the menu items.
   - Click above or below the menu window.
   - Type a closing parens [)] if you've already typed an opening parens [(].
   - Press Escape.

To manually display a code hint:

1. Click in a code location where code hints can appear, such as the following examples:
   - After the dot (.) following a statement or command, where a property or method must be entered
   - Between parentheses [()] in a method name
2. Do one of the following:
   - Click the Show Code Hint button above the Script pane.
   - Press Control+Spacebar (Windows) or Command+Spacebar (Macintosh).
   - If you are working in the Actions panel, open the pop-up menu (at the right side of the title bar), and select Show Code Hint.

Using Escape shortcut keys

You can add many elements to a script by using Escape shortcut keys (pressing the Escape key and then two other keys).

Note: These shortcuts are different from the keyboard shortcuts that initiate certain menu commands.

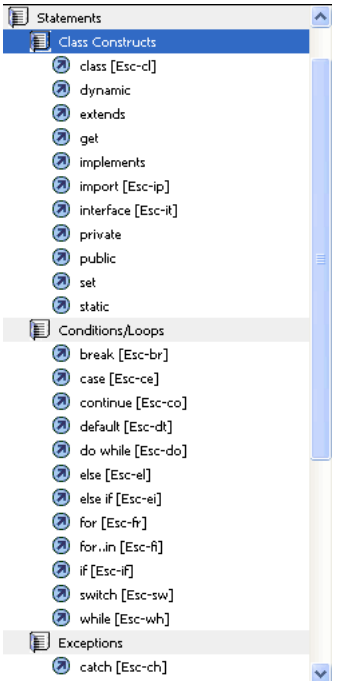
For example, if you are working in the Script pane and type Escape+d+o, the following code is placed in your script, and the insertion point is placed immediately following the word `while`, so you can begin typing your condition:

```
do {
} while ();
```

Similarly, if you type Escape+c+h, the following code is placed in your script, and the insertion point is placed between the parentheses [()], so you can begin typing your condition:

```
catch () {
}
```

If you want to learn (or be reminded) about which commands have Escape shortcut keys, you can show them next to elements in the Actions panel.



To show or hide Escape shortcut keys:

- From the View Options pop-up menu, enable or disable View Escape Shortcut Keys.

Checking syntax and punctuation

To determine whether the code you wrote performs as planned, you need to publish or test the file. However, you can do a quick check of your ActionScript code without leaving the FLA file. Syntax errors are listed in the Output panel. You can also check to see if a set of parentheses, curly braces, or brackets around a block of code is balanced.

When you check syntax, the current script is checked. If the current script calls ActionScript 2.0 classes, those classes are compiled and their syntax is also checked. Other scripts that might be in the FLA file are not checked.

To check syntax:

Do one of the following:

- Click the Check Syntax button above the Script pane.
- In the Actions panel, display the pop-up menu (at the upper right of the panel), and select Check Syntax.
- Press Control+T (Windows) or Command+T (Macintosh).

Note: If you Check Syntax in external ActionScript 2.0 class files, the global class path will affect this process. Sometimes you will generate errors—even if the global class path is set correctly—because the compiler is not aware that this class is being compiled. For more information on compiling classes, see "Compiling and exporting classes" on page 272.

To check for punctuation balance:

1. Click between braces ({}), brackets ([]), or parentheses [()] in your script.
2. For Windows, press Control+' (single quote) or Command+' (Macintosh) to highlight the text between braces, brackets, or parentheses.

   The highlighting helps you check that opening punctuation has corresponding closing punctuation.

Formatting code

You can specify settings to determine if your code is formatted and indented automatically or manually. You can also select whether to view line numbers and whether to wrap long lines of code. In addition, you can select whether to use dynamic font mapping, which ensures that the correct fonts are used when working with multilingual text.

To set format options:

1. Do one of the following

   - In the Actions panel, select Auto Format Options from the Options pop-up menu (at the upper right of the panel).
   - (Flash Professional only) In an external script file, select Edit > Auto Format Options.

   The Auto Format Options dialog box appears.
2. Select any of the check boxes. To see the effect of each selection, look in the Preview pane.

After you set Auto Format Options, your settings are applied automatically to code you write, but not to existing code; you must apply your settings to existing code manually. You might use this procedure to format code that was formatted using different settings, code that you imported from another editor, and so on.

To format code according to Auto Format Options settings:

Do one of the following:

- Click the Auto Format button above the Script pane.
- Select Auto Format from the Actions panel Options pop-up menu.
- Press Control+Shift+F (Windows) or Command+Shift+F (Macintosh).
- In an external script file, select Tools > Auto Format.

To use dynamic font mapping:

- Dynamic font mapping is turned off by default because it increases performance time when scripting. To turn it on, select Use dynamic font mapping in ActionScript preferences.

  If you are working with multilingual text, you should turn on dynamic font mapping, which helps ensure that the correct fonts are used.

To use automatic indentation:

- Automatic indentation is turned on by default. To turn it off, deselect Automatic indentation in ActionScript preferences.

  When automatic indentation is turned on, the text you type after (or { is automatically indented according to the Tab size setting in ActionScript preferences. To indent another line, select the line and press Tab. To remove the indent, select the line and press Shift+Tab.

To enable or disable line numbers and word wrap:

Do one of the following:

- From the View Options pop-up menu, enable or disable View Line Numbers and Word Wrap.
- From the Actions panel Options pop-up menu, enable or disable View Line Numbers and Word Wrap.
- With the pointer focused in the Script pane, press Control+Shift+L (Windows) or Command+Shift+L (Macintosh) to enable or disable word wrap.
- With the pointer focused in the Script pane, press Control+Shift+W (Windows) or Command+Shift+W (Macintosh) to enable or disable line numbers.

Unicode support for ActionScript

Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 support Unicode text encoding for ActionScript. This means that you can include text in different languages in an ActionScript file. For example, you can include text in English, Japanese, and French in the same file.

Caution: When using a non-English application on an English system, the Test Movie command (see "Debugging your scripts" on page 153) fails if any part of the SWF file path has characters that cannot be represented using the Multibyte Character Sets (MBCS) encoding scheme. For example, Japanese paths, which work on a Japanese system, won't work on an English system. All areas of the application that use the external player are subject to this limitation.

You can set ActionScript preferences to specify the type of encoding to use when importing or exporting ActionScript files. You can select UTF-8 encoding or Default Encoding. UTF-8 is 8-bit Unicode format; Default Encoding is the encoding form supported by the language your system is currently using, also called the *traditional code page*.

In general, if you are importing or exporting ActionScript files in UTF-8 format, use the UTF-8 preference. If you are importing or exporting files in the traditional code page in use on your system, use the Default Encoding preference.

If text in your scripts doesn't look as expected when you open or import a file, change the import encoding preference. If you receive a warning message when exporting ActionScript files, you can change the export encoding preference or turn this warning off in ActionScript preferences.

To select text encoding options for importing or exporting ActionScript files:

1. In the Preferences dialog box (Edit > Preferences), click the ActionScript tab.
2. Under Editing Options, do one or both of the following:
   - For Open/Import, select UTF-8 to open or import using Unicode encoding, or select Default Encoding to open or import using the encoding form of the language currently used by your system.
   - For Save/Export, select UTF-8 to save or export using Unicode encoding, or select Default Encoding to save or export using the encoding form of the language currently used by your system.

To turn the export encoding warning off or on:

1. In the Preferences dialog box (Edit > Preferences), click the Warnings tab.
2. Select or deselect Warn on Encoding Conflicts When Exporting .as Files.

Debugging your scripts

Flash provides several tools for testing ActionScript in your SWF files. The Debugger, discussed in the rest of this section, lets you find errors in a SWF file while it's running in Flash Player. Flash also provides the following additional debugging tools:

- The Output panel, which shows error messages (including some runtime errors) and lists of variables and objects (see "Using the Output panel" on page 162)
- The `trace` statement, which sends programming notes and values of expressions to the Output panel (see "Using the trace statement" on page 165)
- The `throw` and `try..catch..finally` statements, which let you test and respond to runtime errors from within your script

You must view your SWF file in a special version of Flash Player called Flash Debug Player. When you install the authoring tool, Flash Debug Player is installed automatically. So, if you install Flash and browse a website that has Flash content, or do a Test Movie, you're using Flash Debug Player. You can also run the installer in the *Flash MX 2004 program*\Players\Debug\ directory or launch the stand-alone Flash Debug Player from the same directory.

When you use the Test Movie command to test SWF files that implement keyboard controls (tabbing, keyboard shortcuts created using `Key.addListener()`, and so on), select Control > Disable Keyboard Shortcuts. Selecting this option prevents the authoring environment from "grabbing" keystrokes, and lets them pass through to the player. For example, in the authoring environment, Control+U opens the Preferences dialog box. If your script assigns Control+U to an action that underlines text onscreen, when you use Test Movie, pressing Control+U opens the Preferences dialog box instead of running the action that underlines text. To let the Control+U command pass through to the player, you must select Control > Disable Keyboard Shortcuts.

Caution: When using a non-English application on an English system, the Test Movie command fails if any part of the SWF file path has characters that cannot be represented using the MBCS encoding scheme. For example, Japanese paths on an English system do not work. All areas of the application that use the external player are subject to this limitation.

The Debugger shows a hierarchical display list of movie clips currently loaded in Flash Player. Using the Debugger, you can display and modify variable and property values as the SWF file plays, and you can use breakpoints to stop the SWF file and step through ActionScript code line by line.

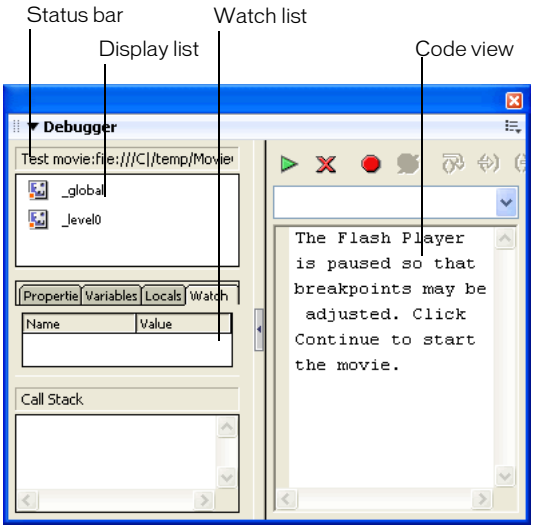
You can use the Debugger in test mode with local files, or you can use it to test files on a web server in a remote location. The Debugger lets you set breakpoints in your ActionScript that stop Flash Player and step through the code as it runs. You can then go back to your scripts and edit them so that they produce the correct results.

After it's activated, the Debugger status bar displays the URL or local path of the file, tells whether the file is running in test mode or from a remote location, and shows a live view of the movie clip display list. When movie clips are added to or removed from the file, the display list reflects the changes immediately. You can resize the display list by moving the horizontal splitter.

To activate the Debugger in test mode:

- Select Control > Debug Movie.

  This command exports the SWF file with debugging information (the SWD file) and enables debugging of the SWF file. It opens the Debugger and opens the SWF file in test mode.



For more information, see the following topics:

- "Debugging a SWF file from a remote location" on page 155
- "Displaying and modifying variables" on page 157
- "Using the Watch list" on page 158
- "Displaying movie clip properties and changing editable properties" on page 158
- "Setting and removing breakpoints" on page 159
- "Stepping through lines of code" on page 160

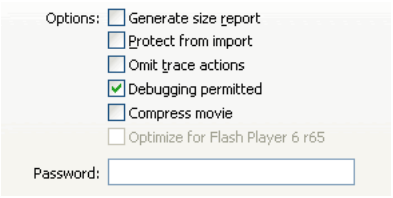
Debugging a SWF file from a remote location

You can debug a remote SWF file using the stand-alone, ActiveX, or plug-in versions of Flash Player. When exporting a SWF file, you can enable debugging in your file and create a debugging password. If you don't enable debugging, the Debugger will not activate.

To ensure that only trusted users can run your SWF files in the Flash Debug Player, you can publish your file with a debugging password. As in JavaScript or HTML, it's possible for users to view client-side variables in ActionScript. To store variables securely, you must send them to a server-side application instead of storing them in your file. However, as a Flash developer, you may have other trade secrets, such as movie clip structures, that you do not want to reveal. You can use a debugging password to protect your work.

To enable remote debugging of a SWF file:

1. Select File > Publish Settings.
2. On the Flash tab of the Publish Settings dialog box, select Debugging permitted.



3. To set a password, enter a password in the Password box.

   After you set this password, no one can download information to the Debugger without the password. However, if you leave the Password box blank, no password is required.
4. Close the Publish Settings dialog box, and select one of the following commands:
   - Control > Debug Movie
   - File > Export Movie
   - File > Publish Settings > Publish

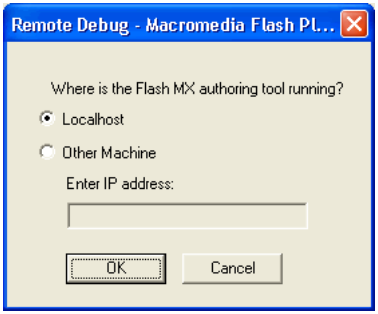
   Flash creates a debugging file, with the extension .swd, and saves it along with the SWF file. The SWD file is used to debug ActionScript, and contains information that lets you use breakpoints and step through code.
5. Place the SWD file in the same directory as the SWF file on the server.

   If the SWD file is not in the same directory as the SWF file, you can still debug remotely; however, the Debugger has no breakpoint information, so you can't step through code.
6. In Flash, select Window > Development Panels > Debugger.
7. In the Debugger, select Enable Remote Debugging from the Options pop-up menu (at the upper right of the panel).

To activate the Debugger from a remote location:

1. Open the Flash authoring application.
2. In a browser or in the debug version of the stand-alone player, open the published SWF file from the remote location.

   The Remote Debug dialog box appears.



   If this dialog box doesn't appear, Flash can't find the SWD file. In this case, right-click (Windows) or Control-click (Macintosh) in the SWF file to display the context menu, and select Debugger.

3. In the Remote Debug dialog box, select Localhost or Other Machine:
   - Select Localhost if the Debug player and the Flash authoring application are on the same computer.
   - Select Other Machine if the Debug player and the Flash authoring application are not on the same computer. Enter the IP address of the computer running the Flash authoring application.
4. When a connection is established, a password prompt appears. Enter your debugging password if you set one.

   The display list of the SWF file appears in the Debugger. If the movie doesn't play, the Debugger might be paused, so click Continue to start it.

Displaying and modifying variables

The Variables tab in the Debugger shows the names and values of any global and Timeline variables in the SWF file selected in the display list. If you change the value of a variable on the Variables tab, you can see the change reflected in the SWF file while it runs. For example, to test collision detection in a game, you can enter the variable value to position a ball in the correct location next to a wall.

The Locals tab in the Debugger shows the names and values of any local variables that are available in the line of ActionScript where the SWF file is currently stopped, at a breakpoint or anywhere else within a user-defined function.

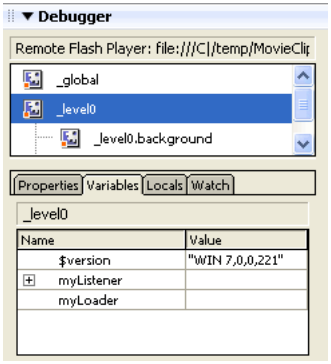
To display a variable:

1. Select the movie clip containing the variable from the display list.

   To display global variables, select the `_global` clip in the display list.

2. Click the Variables tab.

The display list updates automatically as the SWF file plays. If a movie clip is removed from the SWF file at a specific frame, that movie clip, along with its variable and variable name, is also removed from the display list in the Debugger. However, if you mark a variable for the Watch list (see "Using the Watch list" on page 158), the variable is removed from the Variables tab, but can still be viewed in the Watch tab.



To modify a variable value:

- Double-click the value, and enter a new value.

The value cannot be an expression. For example, you can use `"Hello"`, `3523`, or `"http://www.macromedia.com"`, and you cannot use `x + 2` or `eval("name:" +i)`. The value can be a string (any value surrounded by quotation marks [""]), a number, or a Boolean value (`true` or `false`).

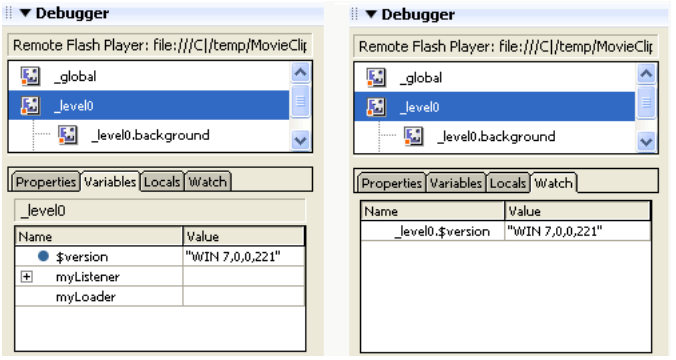
Note: To write the value of an expression to the Output panel in test mode, use the `trace` statement. See "Using the trace statement" on page 165.

Using the Watch list

To monitor a set of critical variables in a manageable way, you can mark variables to appear in the Watch list. The Watch list shows the absolute path to the variable and the value. You can also enter a new variable value in the Watch list the same way as in the Variables tab. The Watch list can show only variables and properties that can be accessed by using an absolute target path, such as _global, _root.

If you add a local variable to the Watch list, its value appears only when Flash Player is stopped at a line of ActionScript where that variable is in scope. All other variables appear while the SWF file is playing. If the Debugger can't find the value of the variable, the value is listed as undefined.

The Watch list can show only variables, not properties or functions.



Variables marked for the Watch list and variables in the Watch list

To add variables to the Watch list:

Do one of the following:

- On the Variables or Locals tab, right-click (Windows) or Control-click (Macintosh) a selected variable and then select Watch from the context menu. A blue dot appears next to the variable.
- On the Watch tab, right-click (Windows) or Control-click (Macintosh) and select Add from the context menu. Double-click in the name column, and enter the target path to the variable name in the field.

To remove variables from the Watch list:

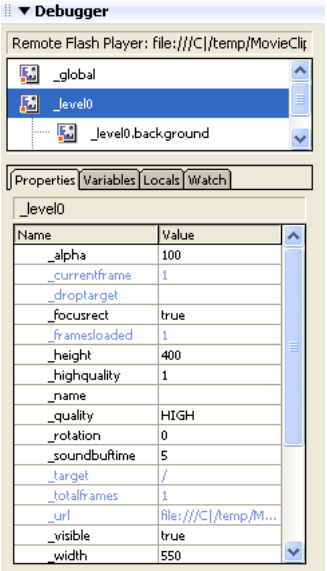
- On the Watch tab, right-click (Windows) or Control-click (Macintosh) and select Remove from the context menu.

Displaying movie clip properties and changing editable properties

The Debugger's Properties tab shows all the property values of any movie clip on the Stage. You can change a value and see its effect in the SWF file while it runs. Some movie clip properties are read-only and cannot be changed.

To display a movie clip's properties in the Debugger:

1. Select a movie clip from the display list.
2. Click the Properties tab in the Debugger.



To modify a property value:

- Double-click the value, and enter a new value.

The value cannot be an expression. For example, you can enter `50` or `"clearwater"`, but you cannot enter `x + 50`. The value can be a string (any value surrounded by quotation marks [""]), a number, or a Boolean value (`true` or `false`). You can't enter object or array values (for example, `{id: "rogue"}` or `[1, 2, 3]`) in the Debugger.

For more information, see "String operators" on page 53 and "Using operators to manipulate values in expressions" on page 49.

Note: To write the value of an expression to the Output panel in test mode, use the `trace` statement. See "Using the trace statement" on page 165.

Setting and removing breakpoints

A breakpoint lets you stop a SWF file running in Flash Player at a specific line of ActionScript. You can use breakpoints to test possible trouble spots in your code. For example, if you've written a set of `if..else if` statements and can't determine which one is executing, you can add a breakpoint before the statements and step through them one by one in the Debugger.

You can set breakpoints in the Actions panel or in the Debugger. (To set breakpoints in external scripts, you must use the Debugger while in a debugging session.) Breakpoints set in the Actions panel are saved with the FLA file. Breakpoints set in the Debugger are not saved in the FLA file and are valid only for the current debugging session.

Caution: If you set breakpoints in the Actions panel and press the Auto Format button, you might notice that some breakpoints are no longer in the correct location. ActionScript might be moved to a different line when your code is formatted because sometimes empty lines are removed. It might be necessary to check and modify your breakpoints after you click Auto Format, or to auto format your scripts before setting breakpoints.

To set or remove a breakpoint in the Actions panel during a debugging session:

Do one of the following:

- Click in the left margin. A red dot indicates a breakpoint.
- 
Click the Debug options button above the Script pane.
- Right-click (Windows) or Control-click (Macintosh) to display the context menu, and select Set Breakpoint, Remove Breakpoint or Remove All Breakpoints.
- Press Control+Shift+B (Windows) or Command+Shift+B (Macintosh).

Note: In previous versions of Flash, clicking in the left margin of the Script pane selected the line of code; now it adds or removes a breakpoint. To select a line of code, use Control-click (Windows) or Command-click (Macintosh).

To set and remove breakpoints in the Debugger:

Do one of the following:

- Click in the left margin. A red dot indicates a breakpoint.
- Click the Toggle Breakpoint or Remove All Breakpoints button above the code view.
- Right-click (Windows) or Control-click (Macintosh) to display the context menu, and select Set Breakpoint, Remove Breakpoint, or Remove All Breakpoints.
- Press Control+Shift+B (Windows) or Command+Shift+B (Macintosh).

After Flash Player stops at a breakpoint, you can step into, over, or out of that line of code.

Note: Do not set breakpoints on comments or empty lines; if breakpoints are set on comments or empty lines, they are ignored.

Stepping through lines of code

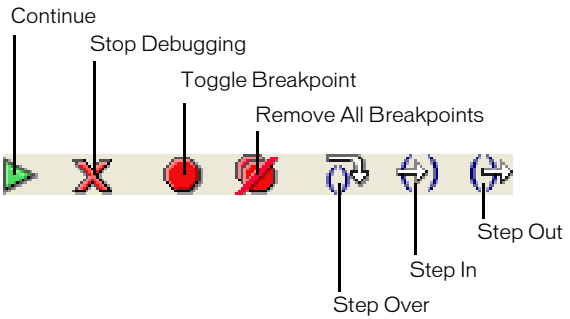
When you start a debugging session, Flash Player is paused so that you can toggle breakpoints. If you set breakpoints in the Actions panel, you can simply click the Continue button to play the SWF file until it reaches a breakpoint. If you didn't set breakpoints in the Actions panel, you can use the jump menu in the Debugger to select any script in the SWF file. When you have selected a script, you can add breakpoints to it.

After adding breakpoints, you must click the Continue button to start the SWF file. The Debugger stops when it reaches the breakpoint. For example, in the following code, suppose a breakpoint is set inside a button on the line `myFunction()`:

```
on(press){
  myFunction();
}
```

When you click the button, the breakpoint is reached and Flash Player pauses. You can now step in to bring the Debugger to the first line of `myFunction()` wherever it is defined in the document. You can also step through or out of the function.

As you step through lines of code, the values of variables and properties change in the Watch list and in the Variables, Locals, and Properties tabs. A yellow arrow on the left side of the Debugger's code view indicates the line at which the Debugger stopped. Use the following buttons along the top of the code view:



Step In advances the Debugger (indicated by the yellow arrow) into a function. Step In works only for user-defined functions.

In the following example, if you place a breakpoint at line 7 and click Step In, the Debugger advances to line 2, and another click of Step In will advance you to line 3. Clicking Step In for lines that do not have user-defined functions in them advances the Debugger over a line of code. For example, if you stop at line 2 and select Step In, the Debugger advances to line 3, as shown in the following example:

```
1 function myFunction() {
2 x = 0;
3 y = 0;
4 }
5
6 mover = 1;
7 myFunction();
8 mover = 0;
```

Step Out advances the Debugger out of a function. This button works only if you are currently stopped in a user-defined function; it moves the yellow arrow to the line after the one where that function was called. In the previous example, if you place a breakpoint at line 3 and click Step Out, the Debugger moves to line 8. Clicking Step Out at a line that is not within a user-defined function is the same as clicking Continue. For example, if you stop at line 6 and click Step Out, the player continues executing the script until it encounters a breakpoint.

Step Over advances the Debugger over a line of code. This button moves the yellow arrow to the next line in the script. In the previous example, if you are stopped at line 7 and click Step Over, you advance directly to line 8 without stepping through `myFunction()`, although the `myFunction()` code still executes.

Continue leaves the line at which the player is stopped and continues playing until a breakpoint is reached.

Stop Debugging makes the Debugger inactive but continues to play the SWF file in Flash Player.

Using the Output panel

In test mode, the Output panel shows information to help you troubleshoot your SWF file. Some information (such as syntax errors) appear automatically. You can show other information by using the List Objects and List Variables commands. (See "Listing a SWF file's objects" on page 163 and "Listing a SWF file's variables" on page 164.)

If you use the `trace` statement in your scripts, you can send specific information to the Output panel as the SWF file runs. This could include notes about the SWF file's status or the value of an expression. (See "Using the trace statement" on page 165.)

To display or hide the Output panel, select Window > Development Panels > Output or press F2.

To work with the contents of the Output panel, click the Options pop-up menu in the upper right corner to see your options.

The following table lists the options available in the Output panel's Options pop-up menu:

Menu item	What it does
Word wrap	Toggles whether long lines will wrap automatically, so the user does not have to use the horizontal scroll bar to view the entire line of characters. If selected, lines will wrap; otherwise, lines will not wrap.
Copy	Copies the entire contents of the Output panel to the computer's Clipboard. To copy a selected portion of the output, select the area you want to copy and then select Copy.
Clear	Clears all output currently in the Output panel.
Find	Opens a dialog box that you can use to find a keyword or phrase within the Output panel contents.
Find Again	Attempts to locate the next instance of a keyword or phrase in the Output panel contents.
Save to File	Saves the current contents of the Output panel to an external text file.

Menu item	What it does
Print	Shows the Print dialog box, which lets you print the current contents of the Output panel to an installed printer or installed programs such as Flash Paper or Acrobat.
Filter level	Lets you select two possible levels of output: None or Verbose. Selecting None suppresses all output sent to the browser.
Maximize Panel	Maximizes the Output panel (when it is docked).
Close Panel	Closes the Output panel and clears the contents of the panel.

For more information on the Output panel, see the following topics:

- "Listing a SWF file's objects" on page 163
- "Listing a SWF file's variables" on page 164
- "Displaying text field properties for debugging" on page 164
- "Using the trace statement" on page 165

Listing a SWF file's objects

In test mode, the List Objects command shows the level, frame, object type (shape, movie clip, or button), target paths, and instance names of movie clips, buttons, and text fields in a hierarchical list. This is especially useful for finding the correct target path and instance name. Unlike the Debugger, the list does not update automatically as the SWF file plays; you must select the List Objects command each time you want to send the information to the Output panel.

Caution: Selecting the List Objects command will clear any information that currently appears in the Output panel. If you do not want to lose information in the Output panel, select Save to File from the Output panel Options pop-up menu or copy and paste the information to another location before selecting the List Objects command.

The List Objects command does not list all ActionScript data objects. In this context, an object is considered to be a shape or symbol on the Stage.

To display a list of objects in a SWF file:

1. If your SWF file is not running in test mode, select Control > Test Movie.
2. Select Debug > List Objects.

A list of all the objects currently on the Stage appears in the Output panel, as shown in the following example:

```
Level #0: Frame=1 Label="Scene_1"
  Button: Target="_level0.myButton"
    Shape:
  Movie Clip: Frame=1 Target="_level0.myMovieClip"
    Shape:
  Edit Text: Target="_level0.myTextField" Text="This is sample text."
```

Listing a SWF file's variables

In test mode, the List Variables command shows a list of all the variables currently in the SWF file. This is especially useful for finding the correct variable target path and variable name. Unlike the Debugger, the list does not update automatically as the SWF file plays; you must select the List Variables command each time you want to send the information to the Output panel.

The List Variables command also shows global variables declared with the `_global` identifier. The global variables appear at the top of the List Variables output in a Global Variables section, and each variable has a `_global` prefix.

In addition, the List Variables command shows getter/setter properties—properties that are created with the `Object.addProperty()` method and invoke `get` or `set` methods. A getter/setter property appears with any other properties in the object to which it belongs. To make these properties easily distinguishable from other variables, the value of a getter/setter property is prefixed with the string `[getter/setter]`. The value that appears for a getter/setter property is determined by evaluating the `get` function of the property.

Caution: Selecting the List Variables command clears any information that appears in the Output panel. If you do not want to lose information in the Output panel, select Save to File from the Output panel Options pop-up menu or copy and paste the information to another location before selecting the List Variables command.

To display a list of variables in a SWF file:

1. If your SWF file is not running in test mode, select Control > Test Movie.
2. Select Debug > List Variables.

   A list of all the variables currently in the SWF file appears in the Output panel, as shown in the following example:

```
Global Variables:
  Variable _global.mycolor = "lime_green"
Level #0:
Variable _level0.$version = "WIN 7,0,19,0"
Variable _level0.myArray = [object #1, class 'Array'] [
    0:"socks",
    1:"gophers",
    2:"mr.claw"
  ]
Movie Clip: Target="_level0.my_mc"
```

Displaying text field properties for debugging

To get debugging information about TextField objects, you can use the Debug > List Variables command in test movie mode. The Output panel uses the following conventions to show TextField objects:

- If a property is not found on the object, it does not appear.
- No more than four properties appear on a line.
- A property with a string value appears on a separate line.
- If there are any other properties defined for the object after the built-in properties are processed, they are added to the display using the rules in the second and third points above.

- Color properties appear as hexadecimal numbers (0x00FF00).
- The properties appear in the following order: `variable`, `text`, `htmlText`, `html`, `textWidth`, `textHeight`, `maxChars`, `borderColor`, `backgroundColor`, `textColor`, `border`, `background`, `wordWrap`, `password`, `multiline`, `selectable`, `scroll`, `hscroll`, `maxscroll`, `maxhscroll`, `bottomScroll`, `type`, `embedFonts`, `restrict`, `length`, `tabIndex`, `autoSize`.

The List Objects command in the Debug menu (during test mode) lists TextField objects. If an instance name is specified for a text field, the Output panel shows the full target path including the instance name in the following form:

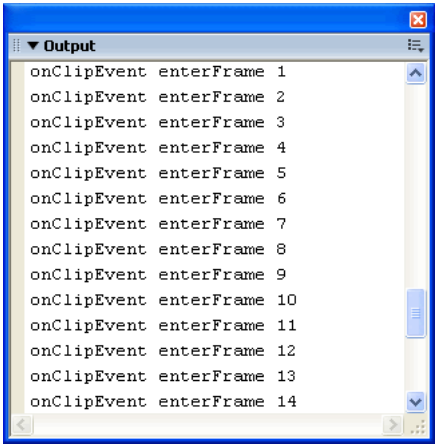
```
Target = "target path"
```

Using the trace statement

When you use the `trace` statement in a script, you can send information to the Output panel. For example, while testing a SWF file or scene, you can send specific programming notes to the panel or have specific results appear when a button is pressed or a frame plays. The `trace` statement is similar to the JavaScript `alert` statement.

When you use the `trace` statement in a script, you can use expressions as parameters. The value of an expression appears in the Output panel in test mode, as shown by the following code snippet and image of the Output panel:

```
onClipEvent (enterFrame) {
  if (i == undefined) {
    i = 0;
  }
  trace("onClipEvent enterFrame "+i++);
}
```



Updating Flash Player for testing

You can download the latest version of Flash Player from the Macromedia Support Center at www.macromedia.com/support/flash and use it to test your SWF files with the most recent version of Flash Player.

CHAPTER 5
Handling Events

An *event* is a software or hardware occurrence that requires a response from a Macromedia Flash application. For example, an event such as a mouse click or a keypress is called a *user event* because it occurs as a result of direct user interaction. An event that is generated automatically by Flash Player, such as the initial appearance of a movie clip on the Stage, is called a *system event* because it isn't generated directly by the user.

In order for your application to react to events, you must use *event handlers*—ActionScript code associated with a particular object and event. For example, when a user clicks a button on the Stage, you might advance the playhead to the next frame. Or when an XML file finishes loading over the network, the contents of that file might appear in a text field.

You can handle events in ActionScript in several ways:

- "Using event handler methods"
- "Using event listeners" on page 169
- "Using button and movie clip event handlers" on page 171, specifically, `on()` and `onClipEvent()`
- "Broadcasting events from component instances" on page 173

Using event handlers with `MovieClip.loadMovie()` can be unpredictable. If you attach an event handler to a button using `on()`, or if you create a dynamic handler using an event handler method such as `MovieClip.onPress`, and then you call `loadMovie()`, the event handler is not available after the new content is loaded. However, if you attach an event handler to a movie clip using `onClipEvent()` or `on()`, and then call `loadMovie()` on that movie clip, the event handler is still available after the new content is loaded.

Using event handler methods

An event handler method is a method of a class that is invoked when an event occurs on an instance of that class. For example, the MovieClip class defines an `onPress` event handler that is invoked whenever the mouse is pressed on a movie clip object. Unlike other methods of a class, however, you don't invoke an event handler directly; Flash Player invokes it automatically when the appropriate event occurs.

The following ActionScript classes define event handlers: “Button class”, “ContextMenu class”, “ContextMenuItem class”, “Key class”, “LoadVars class”, “LocalConnection class”, “Mouse class”, “MovieClip class”, “MovieClipLoader class”, “Selection class”, “SharedObject class”, “Sound class”, “Stage class”, “TextField class”, “XML class”, and “XMLSocket class”. For more information about the event handlers they provide, see each class entry in *Flash ActionScript Language Reference*.

By default, event handler methods are undefined: when a particular event occurs, its corresponding event handler is invoked, but your application doesn't respond further to the event. To have your application respond to the event, you define a function with the function statement and then assign that function to the appropriate event handler. The function you assign to the event handler is then automatically invoked whenever the event occurs.

An event handler consists of three parts: the object to which the event applies, the name of the object's event handler method, and the function you assign to the event handler. The following example shows the basic structure of an event handler:

```
object.eventMethod = function () {
  // Your code here, responding to event
}
```

For example, suppose you have a button named next_btn on the Stage. The following code assigns a function to the button's `onPress` event handler; this function advances the playhead to the next frame in the Timeline:

```
next_btn.onPress = function () {
  nextFrame();
}
```

Assigning a function reference In the previous code, the `nextFrame()` function is assigned directly to `onPress`. You can also assign a function reference (name) to an event handler method and later define the function, as shown in the following example:

```
// Assign a function reference to button's onPress event handler method
next_btn.onPress = goNextFrame;

// Define goNextFrame() function
function goNextFrame() {
  nextFrame();
}
```

Notice in the following example that you assign the function reference, not the function's return value, to the `onPress` event handler:

```
// Incorrect!
next_btn.onPress = goNextFrame();
// Correct.
next_btn.onPress = goNextFrame;
```

Receiving passed parameters Some event handlers receive passed parameters that provide information about the event that occurred. For example, the `TextField.onSetFocus` event handler is invoked when a text field instance gains keyboard focus. This event handler receives a reference to the text field object that previously had keyboard focus.

For example, the following code inserts some text into a text field that no longer has keyboard focus:

```
this.createTextField("my_txt", 99, 10, 10, 200, 20);
my_txt.border = true;
my_txt.type = "input";
this.createTextField("myOther_txt", 100, 10, 50, 200, 20);
myOther_txt.border = true;
myOther_txt.type = "input";
myOther_txt.onSetFocus = function(my_txt:TextField) {
  my_txt.text = "I just lost keyboard focus";
};
```

Event handlers for runtime objects You can also assign functions to event handlers for objects you create at runtime. For example, the following code creates a new movie clip instance (`newclip_mc`) and then assigns a function to the clip's `onPress` event handler:

```
this.attachMovie("symbolID", "newclip_mc", 10);
newclip_mc.onPress = function () {
  trace("You pressed me");
}
```

For more information, see "Creating movie clips at runtime" on page 211.

Overriding event handler methods By creating a class that extends an ActionScript 2.0 class, you can override event handler methods with the functions that you write. You can define an event handler in a new subclass that you can then reuse for various objects by linking any symbol in the library of the extended class to the new subclass. The following code overrides the MovieClip class `onPress` event handler with a function that increases the transparency of the movie clip:

```
// FadeColor class -- fades color when movie clip is clicked
class FadeColor extends MovieClip {
  function onPress() {
    this._alpha = 30;
  }
}
```

For specific instructions on extending an ActionScript 2.0 class and linking to a symbol in the library, see the examples in "Assigning a class to a movie clip symbol" on page 218.

Using event listeners

Event listeners let an object, called a *listener object*, receive events broadcast by another object, called a *broadcaster object*. The broadcaster object registers the listener object to receive events generated by the broadcaster. For example, you can register a movie clip object to receive `onResize` notifications from the Stage, or a button instance could receive `onChanged` notifications from a text field object. You can register multiple listener objects to receive events from a single broadcaster, and you can register a single listener object to receive events from multiple broadcasters.

The listener/broadcaster model for events, unlike event handler methods, lets you have multiple pieces of code listen to the same event without conflict. Event models that do not use the listener/broadcaster model, such as `XML.onLoad`, can be problematic when various pieces of code are listening to the same event; the different pieces of code have conflicts over control of that single XML.onLoad callback function reference. With the listener/broadcaster model, you can easily add listeners to the same event without worrying about code bottlenecks.

The following ActionScript classes can broadcast events: "Key class", "Mouse class", "MovieClipLoader class", "Selection class", "Stage class", and "TextField class". To see which listeners are available for a class, see each class entry.

Event listener model

The event model for event listeners is similar to the model for event handlers (see "Using event handler methods" on page 167), with two main differences:

- You assign the event handler to the listener object, not the object that broadcasts the event.
- You call a special method of the broadcaster object, `addListener()`, which registers the listener object to receive its events.

The following code outlines the event listener model:

```
var listenerObject.eventName = function(){
  // your code here
};
broadcasterObject.addListener(listenerObject);
```

The code starts with an object, *`listenerObject`*, with a property *`eventName`*. Your listener object can be any object, such as an existing object, movie clip, or button instance on the Stage, or it can be an instance of any ActionScript class. For example, a custom movie clip could implement the listener methods for Stage listeners. You could even have one object that listens to several types of listeners.

The *`eventName`* property is an event that occurs on *`broadcasterObject`*, which then broadcasts the event to *`listenerObject`*. You can register multiple listeners to one event broadcaster.

You assign a function to the event listener that responds to the event in some way.

Last, you call the `addListener()` method on the broadcaster object, passing it the name of the listener object.

To unregister a listener object from receiving events, you call the `removeListener()` method of the broadcaster object, passing it the name of the listener object.

```
broadcasterObject.removeListener(listenerObject);
```

Event listener example

The following example shows how to use the `onSetFocus` event listener in the Selection class to create a simple focus manager for a group of input text fields. In this case, the border of the text field that receives keyboard focus is enabled (appears), and the border of the text field that does not have focus is disabled.

To create a simple focus manager with event listeners:

1. Using the Text tool, create a text field on the Stage.
2. Select the text field and, in the Property inspector, select Input from the Text Type pop-up menu, and select the Show Border Around Text option.
3. Create another input text field below the first one.

   Make sure the Show Border Around Text option is not selected for this text field. You can continue to create input text fields.
4. Select Frame 1 in the Timeline and open the Actions panel (Window > Development Panels > Actions).
5. To create an object that listens for focus notification from the Selection class, enter the following code in the Actions panel:

```
// Creates listener object, focusListener
var focusListener:Object = new Object();
// Defines function for listener object
focusListener.onSetFocus = function(oldFocus_txt:TextField,
  newFocus_txt:TextField) {
  oldFocus_txt.border = false;
  newFocus_txt.border = true;
}
```

   This code creates a new (generic) ActionScript object named `focusListener`. This object defines an `onSetFocus` property for itself and assigns a function to the property. The function takes two parameters: a reference to the text field that does not have focus and one to the text field that has focus. The function sets the `border` property of the text field that does not have focus to `false`, and sets the `border` property of the text field that has focus to `true`.
6. To register the `focusListener` object to receive events from the Selection object, add the following code to the Actions panel:

```
// Registers focusListener with broadcaster
Selection.addListener(focusListener);
```

7. Test the application (Control > Test Movie), click in the first text field, and press the Tab key to switch focus between fields.

Using button and movie clip event handlers

You can attach event handlers directly to a button or movie clip instance on the Stage by using the `onClipEvent()` and `on()` handlers. The `onClipEvent()` handler broadcasts movie clip events, and the `on()` handler handles button events.

To attach an event handler to a button or movie clip instance, click the button or movie clip instance on the Stage to bring it in focus, and then enter code in the Actions Panel. The title of the Actions panel reflects that code will be attached to the button or movie clip: Actions Panel - Button or Actions Panel - Movie Clip. For guidelines about using code that's attached to button or movie clip instances, see Chapter 3, "Organizing ActionScript in a document," on page 82.

Note: Do not confuse button and movie clip event handlers with component events, such as `SimpleButton.click`, `UIObject.hide`, and `UIObject.reveal`, which must be attached to component instances and are discussed in Using Components Help.

You can attach `onClipEvent()` and `on()` only to movie clip instances that have been placed on the Stage during authoring. You cannot attach `onClipEvent()` or `on()` to movie clip instances that are created at runtime (using the `attachMovie()` method, for example). To attach event handlers to objects created at runtime, use event handler methods or event listeners. (See "Using event handler methods" on page 167 and "Using event listeners" on page 169.)

For more information on button and movie clip event handlers, see the following topics:

- "Specifying events for on() or onClipEvent()" on page 172
- "Attaching multiple handlers to one object" on page 172
- "Using on() and onClipEvent() with event handler methods" on page 173

Specifying events for on() or onClipEvent()

To use an `on()` or `onClipEvent()` handler, attach it directly to an instance of a button or movie clip on the Stage and specify the event you want to handle for that instance. For a complete list of events supported by the `on()` and `onClipEvent()` event handlers, see `on()` and `onClipEvent()` in *Flash ActionScript Language Reference*.

For example, the following `on()` event handler executes whenever the user clicks the button to which the handler is attached:

```
on(press) {
  trace("Thanks for pressing me.");
}
```

You can specify two or more events for each `on()` handler, separated by commas. The ActionScript in a handler executes when either of the events specified by the handler occurs. For example, the following `on()` handler attached to a button executes whenever the mouse rolls over and then off the button:

```
on(rollOver, rollOut) {
  trace("You rolled over, or rolled out");
}
```

Attaching multiple handlers to one object

You can also attach more than one handler to an object if you want different scripts to run when different events occur. For example, you could attach the following `onClipEvent()` handlers to the same movie clip instance. The first executes when the movie clip first loads (or appears on the Stage); the second executes when the movie clip is unloaded from the Stage.

```
onClipEvent(load) {
  trace("I've loaded");
}
onClipEvent (unload) {
  trace("I've unloaded");
}
```

Using on() and onClipEvent() with event handler methods

You can, in some cases, use different techniques to handle events without conflict. Using `on()` and `onClipEvent()` doesn't conflict with using event handler methods that you define.

For example, suppose you have a button in a SWF file; the button can have an `on(press)` handler that tells the SWF file to play, and the same button can have an `onPress` method, for which you define a function that tells an object on the Stage to rotate. When you click the button, the SWF file plays and the object rotates. Depending on when and what kinds of events you want to invoke, you can use `on()` and `onClipEvent()`, event handler methods, or both techniques of event handling.

However, the scope of variables and objects in `on()` and `onClipEvent()` handlers is different than in event handler and event listeners. See "Event handler scope" on page 174.

You can also use `on()` with movie clips to create movie clips that receive button events. For more information, see "Creating movie clips with button states" on page 173.

Broadcasting events from component instances

For any component instance, you can specify how an event is handled. Component events are handled differently than events broadcast from native ActionScript objects.

For more information, see "Handling Component Events" in *Using Components*.

Creating movie clips with button states

When you attach an `on()` handler to a movie clip, or assign a function to one of the MovieClip mouse event handlers for a movie clip instance, the movie clip responds to mouse events in the same way as a button. You can also create automatic button states (Up, Over, and Down) in a movie clip by adding the frame labels `_up`, `_over`, and `_down` to the movie clip's Timeline.

When the user moves the mouse over the movie clip or clicks it, the playhead is sent to the frame with the appropriate frame label. To designate the hit area used by a movie clip, you use the `MovieClip.hitArea` property.

To create button states in a movie clip:

1. Select a frame in a movie clip's Timeline to use as a button state (Up, Over, or Down).
2. Enter a frame label in the Property inspector (`_up`, `_over`, or `_down`).
3. To add additional button states, repeat steps 1–2.
4. To make the movie clip respond to mouse events, do one of the following:
   - Attach an `on()` event handler to the movie clip instance, as discussed in "Using button and movie clip event handlers" on page 171).
   - Assign a function to one of the movie clip object's mouse event handlers (`onPress`, `onRelease`, and so forth), as discussed in "Using event handler methods" on page 167.

Event handler scope

The scope, or *context*, of variables and commands that you declare and execute within an event handler depends on the type of event handler you use: event handlers or event listeners, or `on()` and `onClipEvent()` handlers. If you're defining an event handler in a new ActionScript 2.0 class, the scope also depends on how you define the event handler. This section contains both ActionScript 1 and ActionScript 2.0 examples.

ActionScript 1 examples Functions assigned to event handler methods and event listeners (as with all ActionScript functions that you write) define a local variable scope, but `on()` and `onClipEvent()` handlers do not.

For example, consider the following two event handlers. The first is an `onPress` event handler associated with a movie clip named `clip_mc`. The second is an `on()` handler attached to the same movie clip instance.

```
// Attached to clip_mc's parent clip Timeline:
clip_mc.onPress = function () {
  var color; // local function variable
  color = "blue";
}
// on() handler attached to clip_mc:
on(press) {
  var color; // no local variable scope
  color = "blue";
}
```

Although both event handlers contain the same code, they have different results. In the first case, the `color` variable is local to the function defined for `onPress`. In the second case, because the `on()` handler doesn't define a local variable scope, the variable is defined in the scope of the Timeline of the movie clip `clip_mc`.

For `on()` event handlers attached to buttons, rather than to movie clips, variables (as well as function and method calls) are invoked in the scope of the Timeline that contains the button instance.

For instance, the following `on()` event handler produces different results that depend on whether it's attached to a movie clip or button object. In the first case, the `play()` function call starts the playhead of the Timeline that contains the button; in the second case, the `play()` function call starts the Timeline of the movie clip to which the handler is attached.

```
// Attached to button
on(press) {
  play(); // plays parent Timeline
}
// Attached to movie clip
on(press) {
  play(); // plays movie clip's Timeline
}
```

When attached to a button object, the `play()` function applies to the Timeline that contains the button—that is, the button's parent Timeline. But when the `on(press)` handler is attached to a movie clip object, the `play()` function call applies to the movie clip that bears the handler. If you attach the following code to a movie clip, it plays the parent Timeline:

```
// Attached to movie clip
on(press) {
  _parent.play(); // plays parent Timeline
}
```

Within an event handler or event listener definition, the same `play()` function would apply to the Timeline that contains the function definition. For example, suppose you declare the following `my_mc.onPress` event handler method on the Timeline that contains the movie clip instance `my_mc`:

```
// Function defined on a Timeline:
my_mc.onPress = function() {
  play(); //plays Timeline that it is defined on
};
```

If you want to play the movie clip that defines the `onPress` event handler, you have to refer explicitly to that clip using the `this` keyword, as follows:

```
// Function defined on root Timeline
my_mc.onPress = function () {
  this.play(); // plays Timeline of my_mc clip
};
```

However, the same code placed on the root Timeline for a button instance would instead play the root Timeline:

```
my_btn.onPress = function() {
  this.play(); //plays root Timeline
};
```

For more information about the scope of the `this` keyword in event handlers, see "Scope of the this keyword" on page 176.

ActionScript 2.0 example The following Loader class is used to load a text file and returns a variable after it successfully loads the file.

```
class Loader {
  var params_lv:LoadVars;
  function Loader() {
    params_lvL:LoadVars = new LoadVars();
    params_lv.onLoad = onLoadVarsDone; // <-- this is a problem!
    params_lv.load("foo.txt");
  }
  function onLoadVarsDone(success:Boolean):Void {
    trace(params_lv.someVariable);
  }
}
```

This code cannot work correctly because there is a problem involving scope with the event handlers, and what `this` refers to is confused between the `onLoad` event handler and the class. The behavior that you might expect in this example is that the `onLoadVarsDone()` method will be invoked in the scope of the Loader object; but it is invoked in the scope of the LoadVars object because the method has been extracted from the Loader object and grafted onto the LoadVars object. The LoadVars object then invokes the `this.onLoad` event handler when the text file has successfully loaded, and the `onLoadVarsDone()` function is invoked with `this` set to LoadVars, not Loader. The `params_lv` object resides in the this scope when it is invoked, even though the `onLoadVarsDone()` function relies on the `params_lv` object by reference. Therefore, it is expecting a `params_lv.params_lv` instance that does not exist.

To correctly invoke the `onLoadVarsDone()` method in the scope of the Loader object, you can use the following strategy: Use a function literal to create an anonymous function that calls the desired function. The `owner` object is still visible in the scope of the anonymous function, so it can be used to find the calling Loader object.

```
class Loader {
  var params_lv:LoadVars;
  function Loader() {
    params_lv:LoadVars = new LoadVars();
    var owner:Loader = this;
    params_lv.onLoad = function (success:Boolean) {
  owner.onLoadVarsDone(success); }
    params_lv.load("foo.txt");
  }
  function onLoadVarsDone(success:Boolean):Void {
    trace(params_lv.someVariable);
  }
}
```

Scope of the this keyword

The `this` keyword refers to the object in the currently executing scope. Depending on what type of event handler technique you use, `this` can refer to different objects.

Within an event handler or event listener function, `this` refers to the object that defines the event handler or event listener method. For example, in the following code, `this` refers to `my_mc`:

```
// onPress() event handler attached to _level0.my_mc:
my_mc.onPress = function () {
  trace(this); // displays '_level0.my_mc'
}
```

Within an `on()` handler attached to a movie clip, `this` refers to the movie clip to which the `on()` handler is attached, as shown in the following code:

```
// Attached to movie clip named my_mc
on(press) {
  trace(this); displays '_level0.my_mc'
}
```

Within an `on()` handler attached to a button, `this` refers to the Timeline that contains the button, as shown in the following code:

```
// Attached to button on main Timeline
on(press) {
  trace(this); // displays '_level0'
}
```

CHAPTER 6
Creating Interaction with ActionScript

In simple animations, Macromedia Flash Player plays the scenes and frames of a SWF file sequentially. In an interactive SWF file, your audience uses the keyboard and mouse to jump to different parts of a SWF file, move objects, enter information in forms, and perform many other interactive operations.

You use ActionScript to create scripts that tell Flash Player what action to perform when an event occurs. Some events that can trigger a script occur when the playhead reaches a frame, when a movie clip loads or unloads, or when the user clicks a button or presses a key.

A script can consist of a single command, such as instructing a SWF file to stop playing, or a series of commands and statements, such as first evaluating a condition and then performing an action. Many ActionScript commands are simple and let you create basic controls for a SWF file. Other actions require some familiarity with programming languages and are intended for advanced development.

About events and interaction

Whenever a user clicks the mouse or presses a key, that action generates an event. These types of events are generally called *user events* because they are generated in response to some action by the user. You can write ActionScript to respond to, or *handle*, these events. For example, when a user clicks a button, you might want to send the playhead to another frame in the SWF file or load a new web page into the browser.

In a SWF file, buttons, movie clips, and text fields all generate events to which you can respond. ActionScript provides three ways to handle events: event handler methods, event listeners, and `on()` and `onClipEvent()` handlers. For more information about events and handling events, see Chapter 5, “Handling Events,” on page 167.

Controlling SWF file playback

The following ActionScript functions let you control the playhead in the Timeline and load a new web page into a browser window:

- The `gotoAndPlay()` and `gotoAndStop()` functions send the playhead to a frame or scene. These are global functions that you can call from any script. You can also use the `MovieClip.gotoAndPlay()` and `MovieClip.gotoAndStop()` methods to navigate the Timeline of a specific movie clip object.
- The `play()` and `stop()` actions play and stop movies.
- The `getURL()` action jumps to a different URL.

Jumping to a frame or scene

To jump to a specific frame or scene in the SWF file, you can use the `gotoAndPlay()` and `gotoAndStop()` global functions or the equivalent `MovieClip.gotoAndPlay()` and `MovieClip.gotoAndStop()` methods of the MovieClip class. Each function or method lets you specify a frame to jump to in the current scene. If your document contains multiple scenes, you can specify a scene and frame where you want to jump.

The following example uses the global `gotoAndPlay()` function within a button object's `onRelease` event handler to send the playhead of the Timeline that contains the button to Frame 10:

```
jump_btn.onRelease = function () {
  gotoAndPlay(10);
}
```

In the next example, the `MovieClip.gotoAndStop()` method sends the Timeline of a movie clip instance named `categories_mc` to Frame 10 and stops. When you use the MovieClip methods `gotoAndPlay()` and `gotoAndStop()`, you must specify an instance to which the method applies.

```
jump_btn.onPress = function () {
  categories_mc.gotoAndStop(10);
}
```

Playing and stopping movie clips

Unless instructed otherwise, after a SWF file starts, it plays through every frame in the Timeline. You can start or stop a SWF file by using the `play()` and `stop()` global functions or the equivalent MovieClip methods. For example, you can use `stop()` to stop a SWF file at the end of a scene before proceeding to the next scene. After a SWF file stops, it must be explicitly started again by calling `play()` or `gotoAndPlay()`.

You can use the `play()` and `stop()` functions or MovieClip methods to control the main Timeline or the Timeline of any movie clip or loaded SWF file. The movie clip you want to control must have an instance name and must be present in the Timeline.

The following `on(press)` handler attached to a button starts the playhead moving in the SWF file or movie clip that contains the button object:

```
// Attached to a button instance
on(press) {
```

```
  // Plays the Timeline that contains the button
  play();
}
```

This same `on()` event handler code produces a different result when attached to a movie clip object rather than a button. When attached to a button object, statements made within an `on()` handler are applied to the Timeline that contains the button, by default. However, when attached to a movie clip object, statements made within an `on()` handler are applied to the movie clip to which the `on()` handler is attached.

For example, the following `onPress()` handler code stops the Timeline of the movie clip to which the handler is attached, not the Timeline that contains the movie clip:

```
//Attached to the myMovie_mc movie clip instance
myMovie_mc.onPress() {
  stop();
}
```

The same conditions apply to `onClipEvent()` handlers attached to movie clip objects. For example, the following code stops the Timeline of the movie clip that bears the `onClipEvent()` handler when the clip first loads or appears on the Stage:

```
onClipEvent(load) {
  stop();
}
```

Jumping to a different URL

To open a web page in a browser window, or to pass data to another application at a defined URL, you can use the `getURL()` global function or the `MovieClip.getURL()` method. For example, you can have a button that links to a new website, or you can send Timeline variables to a CGI script for processing in the same way as you would an HTML form. You can also specify a target window, the same as you would when targeting a window with an HTML anchor (`<a></a>`) tag.

For example, the following code opens the macromedia.com home page in a blank browser window when the user clicks the button instance named `homepage_btn`:

```
//Attach to frame
homepage_btn.onRelease = function () {
  getURL("http://www.macromedia.com", "_blank");
};
```

You can also send variables along with the URL, using `GET` or `POST` methods. This is useful if the page you are loading from an application server, such as a ColdFusion Server (CFM) page, expects to receive form variables. For example, suppose you want to load a CFM page named addUser.cfm that expects two form variables, `name` and `age`. To do this, you can create a movie clip named `variables_mc` that defines those two variables, as shown in the following example:

```
variables_mc.name = "Francois";
variables_mc.age = 32;
```

The following code then loads addUser.cfm into a blank browser window and passes to the CFM page `variables_mc.name` and `variables_mc.age` in the `POST` header:

```
variables_mc.getURL("addUser.cfm", "_blank", "POST");
```

For more information, see `getURL()` in *Flash ActionScript Language Reference*.

Creating interactivity and visual effects

To create interactivity and other visual effects, you need to understand the following techniques:

Creating a custom mouse pointer

A standard mouse pointer is the operating system's onscreen representation of the position of the user's mouse. By replacing the standard pointer with one you design in Flash, you can integrate the user's mouse movement within the SWF file more closely. The sample in this section uses a custom pointer that looks like a large arrow. The power of this feature, however, is your ability to make the custom pointer look like anything—for example, a football to be carried to the goal line or a swatch of fabric pulled over a chair to change its color.

To create a custom pointer, you design the pointer movie clip on the Stage. Then, in ActionScript, you hide the standard pointer and track its movement. To hide the standard pointer, you use the `Mouse.hide()` method of the built-in Mouse class.

To create a custom pointer:

1. Create a movie clip to use as a custom pointer, and place an instance of the clip on the Stage.
2. Select the movie clip instance on the Stage.
3. In the Property inspector, type `cursor_mc` into the instance name text box.
4. Select Window > Development Panels > Actions to open the Actions panel if it is not already visible.
5. Type the following code in the Actions panel:

```
Mouse.hide();
cursor_mc.onMouseMove = function() {
  this._x = _xmouse;
  this._y = _ymouse;
  updateAfterEvent();
};
```

The `Mouse.hide()` method hides the pointer when the movie clip first appears on the Stage; the `onMouseMove` function positions the custom pointer at the same place as the pointer and calls `updateAfterEvent` whenever the user moves the mouse.

The `updateAfterEvent` function immediately refreshes the screen after the specified event occurs, rather than when the next frame is drawn, which is the default behavior. For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help.

6. Select Control > Test Movie to test your custom pointer.

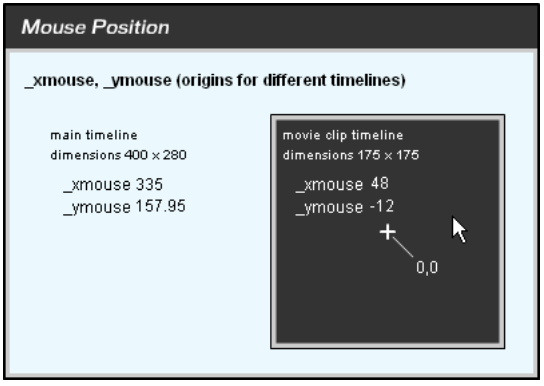
Buttons still function when you use a custom pointer. It's a good idea to put the custom pointer on the top layer of the Timeline so that, as you move the mouse in the SWF file, the custom pointer appears in front of buttons and other objects in other layers. Also, the tip of a custom pointer is the registration point of the movie clip you're using as the custom pointer. Therefore, if you want a certain part of the movie clip to act as the tip of the pointer, set the registration point coordinates of the clip to be that point.

For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help.

Getting the pointer position

You can use the `_xmouse` and `_ymouse` properties to find the location of the pointer in a SWF file. These properties could be used, for example, in a map application that gets the value of the `_xmouse` and `_ymouse` properties and uses them to calculate the longitude and latitude of a specific location.

Each Timeline has an `_xmouse` and `_ymouse` property that returns the location of the pointer within its coordinate system. The position is always relative to the registration point. For the main Timeline (`_level0`), the registration point is the upper left corner. For a movie clip, the registration point depends on the registration point set when the clip was created or its placement on the Stage.



The _xmouse and _ymouse properties within the main Timeline and a movie clip Timeline

The following procedure shows several ways to get the pointer position within the main Timeline or within a movie clip.

To get the current pointer position:

1. Create two dynamic text fields, and name them box1_txt and box2_txt.
2. Add labels for the text boxes: X position and Y position, respectively.
3. Select Window > Development Panels > Actions to open the Actions panel if it is not already open.
4. Add only one of the following blocks of code to the script pane:

```
var mouseListener:Object = new Object();
mouseListener.onMouseMove = function() {
  //returns the X and Y position of the mouse
  box1_txt.text = _xmouse;
  box2_txt.text = _ymouse;
};
Mouse.addListener(mouseListener);
```

5. Select Control > Test Movie to test the movie. The box1_txt and box2_txt fields show the position of the pointer while you move it over the Stage.

For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help.. For more information about the `_xmouse` and `_ymouse` properties, see `MovieClip._xmouse` and `MovieClip._ymouse` in ActionScript Dictionary Help.

Capturing keypresses

For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help.See the *`keyPress`* parameter of the `on()` handler entry in ActionScript Language Reference Help.

You can use the methods of the built-in Key class to detect the last key pressed by the user. The Key class does not require a constructor function; to use its methods, you simply call the methods on the class, as shown in the following example:

```
Key.getCode();
```

You can obtain either virtual key codes or American Standard Code for Information Interchange (ASCII) values of keypresses:

- To obtain the virtual key code of the last key pressed, use the `getCode()` method.
- To obtain the ASCII value of the last key pressed, use the `getAscii()` method.

A virtual key code is assigned to every physical key on a keyboard. For example, the left arrow key has the virtual key code 37. By using a virtual key code, you ensure that your SWF file's controls are the same on every keyboard, regardless of language or platform.

ASCII values are assigned to the first 127 characters in every character set. ASCII values provide information about a character on the screen. For example, the letter "A" and the letter "a" have different ASCII values.

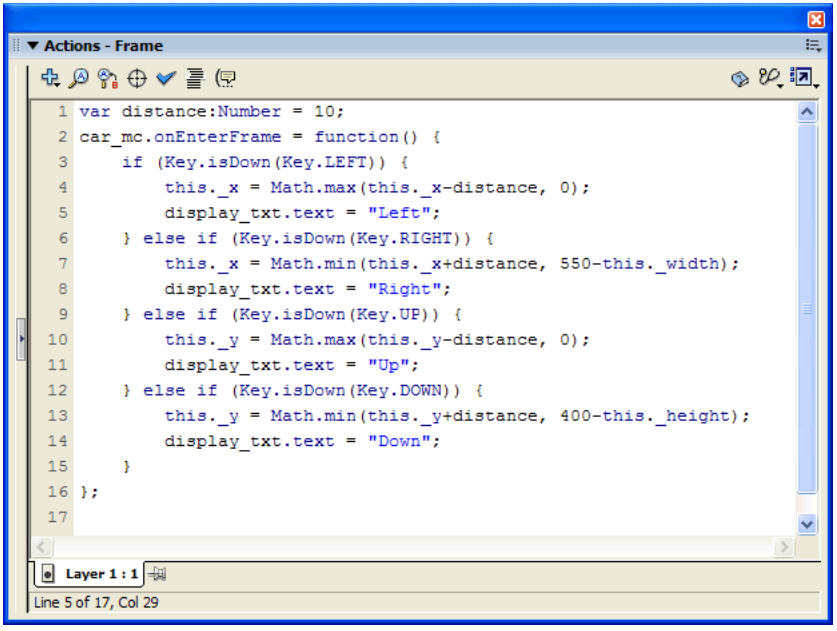
To decide which keys to use and determine their virtual key codes, use one of the following approaches:

- See the list of key codes in Appendix C, "Keyboard Keys and Key Code Values," on page 313. See the list of key codes in Appendix C, "Keyboard Keys and Key Code Values," in ActionScript Dictionary Help.
- Use a Key class constant. (In the Actions toolbox, click Built-in Classes > Movie > Key > Constants.)
- Assign the following `onClipEvent()` handler to a movie clip, and select Control > Test Movie and press the desired key:

```
onClipEvent(keyDown) {
  trace(Key.getCode());
}
```

The key code of the desired key appears in the Output panel.

A common place to use Key class methods is within an event handler. In the following example, the user moves the car using the arrow keys. The `Key.isDown()` method indicates whether the key being pressed is the right, left, up, or down arrow. The event handler, `onEnterFrame`, determines the `Key.isDown(keyCode)` value from the `if` statements. Depending on the value, the handler instructs Flash Player to update the position of the car and to show the direction.



The input from the keyboard keys moves the car.

The following procedure shows how to capture keypresses to move a movie clip up, down, left, or right on the Stage, depending on which corresponding arrow key (up, down, left, or right) is pressed. The movie clip is confined to an arbitrary area that is 400 pixels wide and 300 pixels high. Also, a text field shows the name of the pressed key.

To create a keyboard-activated movie clip:

1. On the Stage, create a movie clip that can move in response to keyboard arrow activity.

   In this example, the movie clip instance name is `car`.

2. On the Stage, create a dynamic text box that can update with the direction of the car. Using the Property inspector, give it an instance name of `display_txt`.

   Note: Don't confuse variable names with instance names. For more information, see "About text field instance and variable names" on page 222.

3. Select Frame 1 in the Timeline; then select Window > Development Panels > Actions to open the Actions panel if it is not already visible.

4. To set how far the car moves across the screen with each keypress, define a `distance` variable and set its initial value to 10:

   ```
   var distance:Number = 10;
   ```

5. To create the event handler for the car movie clip that checks which arrow key (Left, Right, Up, or Down) is currently pressed, add the following code to the Actions panel:

   ```
   car_mc.onEnterFrame = function() {

   };
   ```

6. To check if the Left Arrow key is pressed and to move the car movie clip accordingly, add code to the body of the `onEnterFrame` event handler. Your code should look like the following example (new code is in bold):

   ```
   var distance:Number = 10;
   car_mc.onEnterFrame = function() {
     if (Key.isDown(Key.LEFT)) {
       this._x = Math.max(this._x-distance, 0);
       display_txt.text = "Left";
     }
   };
   ```

   If the Left Arrow key is pressed, the car's _x property is set to the current _x value minus distance or the value 0, whichever is greater. Therefore, the value of the _x property can never be less than 0. Also, the word *Left* should appear in the SWF file.

7. Use similar code to check if the Right, Up, or Down arrow key is being pressed. Your complete code should look like the following example (new code is in bold):

   ```
   var distance:Number = 10;
   car_mc.onEnterFrame = function() {
     if (Key.isDown(Key.LEFT)) {
       this._x = Math.max(this._x-distance, 0);
       display_txt.text = "Left";
     } else if (Key.isDown(Key.RIGHT)) {
       this._x = Math.min(this._x+distance, 550-this._width);
   ```

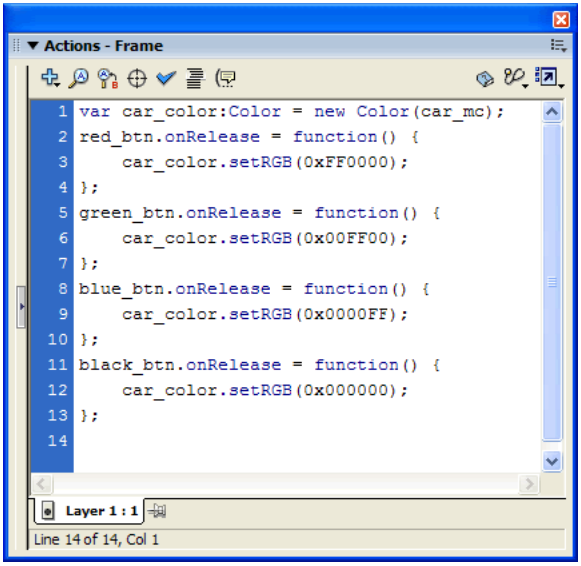
```
        display_txt.text = "Right";
    } else if (Key.isDown(Key.UP)) {
        this._y = Math.max(this._y-distance, 0);
        display_txt.text = "Up";
    } else if (Key.isDown(Key.DOWN)) {
        this._y = Math.min(this._y+distance, 400-this._height);
        display_txt.text = "Down";
    }
};
```

8. Select Control > Test Movie to test the file.

For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help. For more information about the methods of the Key class, see the Key class entry in ActionScript Dictionary Help.

Setting color values

You can use the methods of the built-in Color class to adjust the color of a movie clip. The `setRGB()` method assigns hexadecimal red, green, blue (RGB) values to the movie clip. The following example uses `setRGB()` to change an object's color based on user input.



A button action creates a Color object and changes the color of the car based on user input.

To set the color value of a movie clip:

1. Select a movie clip on the Stage.
2. In the Property inspector, enter `carColor` as the instance name.
3. Create a button named colorChip, place four instances of the button on the Stage, and name them red_btn, green_btn, blue_btn, and black_btn.
4. Select Frame 1 in the main Timeline, and select Window > Development Panels > Actions.

5. To create a Color object that targets the `car_mc` movie clip, add the following code to the Actions panel:

```
var myColor_color:Color = new Color(car_mc);
```

6. To make the red button change the color of the `carColor` movie clip to red, add the following code to the Actions panel:

```
red_btn.onRelease = function() {
  car_color.setRGB(0xFF0000);
};
```

   The hexadecimal value `0xFF0000` is red.

7. Repeat step 6 for the other buttons (green_btn, blue_btn, and black_btn) to change the color of the movie clip to the corresponding color. Your code should now look like the following example (new code is in bold):

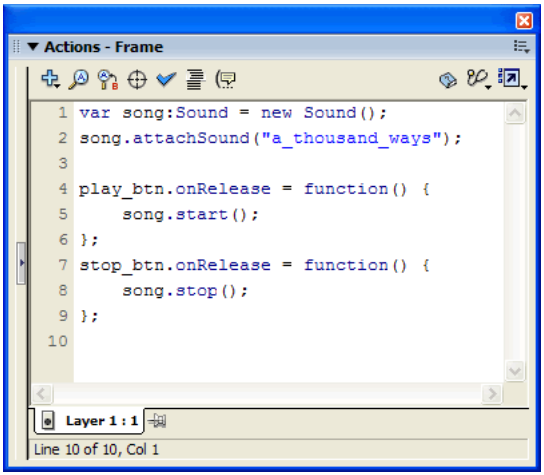
```
var car_color:Color = new Color(car_mc);
red_btn.onRelease = function() {
  car_color.setRGB(0xFF0000);
};
green_btn.onRelease = function() {
  car_color.setRGB(0x00FF00);
};
blue_btn.onRelease = function() {
  car_color.setRGB(0x0000FF);
};
black_btn.onRelease = function() {
  car_color.setRGB(0x000000);
};
```

8. Select Control > Test Movie to change the color of the movie clip.

For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help.

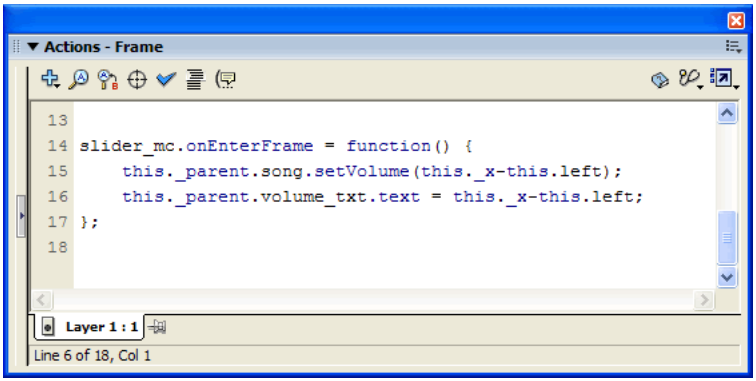
Creating sound controls

You use the built-in Sound class to control sounds in a SWF file. To use the methods of the Sound class, you must first create a Sound object. Then you can use the `attachSound()` method to insert a sound from the library into a SWF file while the SWF file is running.



When the user releases the Play button, a song plays through the speaker.

The Sound class's `setVolume()` method controls the volume, and the `setPan()` method adjusts the left and right balance of a sound.



When the user drags the volume slider, the setVolume() method is called.

The following procedures show how to create sound controls similar to the ones shown in the figure.

To attach a sound to a Timeline:

1. Select File > Import to import a sound.
2. Select the sound in the library, right-click (Windows) or Control-click (Macintosh), and select Linkage.
3. Select Export for ActionScript and Export in First Frame; then give it the identifier `a_thousand_ways`.
4. Add a button to the Stage and name it `play_btn`.
5. Add a button to the Stage and name it `stop_btn`.
6. Select Frame 1 in the main Timeline, and select Window > Development Panels > Actions. Add the following code to the Actions panel:

```
var song:Sound = new Sound();
song.attachSound("a_thousand_ways");
play_btn.onRelease = function() {
  song.start();
};
stop_btn.onRelease = function() {
  song.stop();
};
```

   This code first stops the speaker movie clip. It then creates a new Sound object (`song`) and attaches the sound whose linkage identifier is `a_thousand_ways`. The `onRelease` event handlers associated with the `playButton` and `stopButton` objects start and stop the sound using the `Sound.start()` and `Sound.stop()` methods, and also play and stop the attached sound.
7. Select Control > Test Movie to hear the sound.

To create a sliding volume control:

1. Drag a button to the Stage, and enter `knob_btn` as the instance name in the Property inspector.
2. Select the button, and select Modify > Convert to Symbol. Be careful to select the movie clip behavior.

   This creates a movie clip with the button on its first frame.
3. Select the movie clip, and enter `slider_mc` as the instance name in the Property inspector.
4. Select Frame 1 of the main Timeline, and select Window > Development Panels > Actions.
5. Enter the following code into the Actions panel:

```
slider_mc.top = slider_mc._y;
slider_mc.bottom = slider_mc._y;
slider_mc.left = slider_mc._x;
slider_mc.right = slider_mc._x+100;
slider_mc._x += 100;
```

6. Enter the following event handlers:

```
slider_mc.knob_btn.onPress = function() {
  startDrag(this._parent, false, this._parent.left, this._parent.top,
  this._parent.right, this._parent.bottom);
};
slider_mc.knob_btn.onRelease = function() {
  stopDrag();
};
slider_mc.onEnterFrame = function() {
  this._parent.song.setVolume(this._x-this.left);
  this._parent.volume_txt.text = this._x-this.left;
};
```

The `startDrag()` parameters `left`, `top`, `right`, and `bottom` are variables set in a clip action.

7. Select Control > Test Movie to use the volume slider.

To create a sliding balance control:

1. Drag a button to the Stage, and enter `knob_btn` as the instance name in the Property inspector.
2. Select the button, and select Insert > Convert to Symbol. Select the movie clip property.
3. Select the movie clip, and enter `slider_mc` as the instance name in the Property inspector.
4. Select the button, and select Window > Development Panels > Actions.
5. Enter the following code into the Actions panel:

```
slider_mc.top = slider_mc._y;
slider_mc.bottom = slider_mc._y;
slider_mc.left = slider_mc._x;
slider_mc.right = slider_mc._x+100;
slider_mc._x += 100;

slider_mc.onEnterFrame = function() {
  if (this.dragging) {
    trace((this._x-this.center)*2);
  }
};

slider_mc.knob_btn.onPress = function() {
  startDrag(this._parent, false, this._parent.left, this._parent.top,
  this._parent.right, this._parent.bottom);
  dragging = true;
};
slider_mc.knob_btn.onRelease = function() {
  stopDrag();
  dragging = false;
};
```

The `startDrag()` parameters `left`, `top`, `right`, and `bottom` are variables set in a clip action.

6. Select Control > Test Movie to use the balance slider.

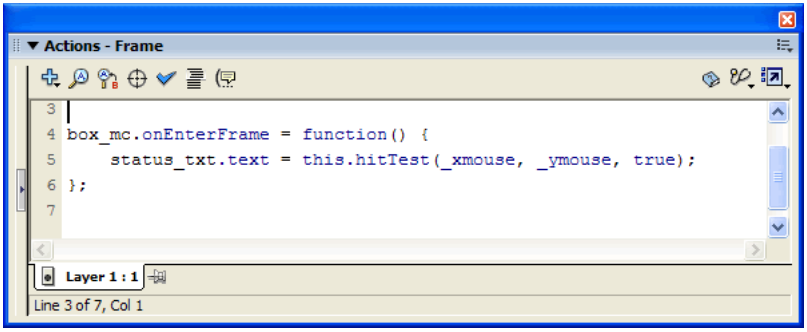
For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help.

Detecting collisions

The `hitTest()` method of the MovieClip class detects collisions in a SWF file. It checks to see if an object has collided with a movie clip and returns a Boolean value (`true` or `false`).

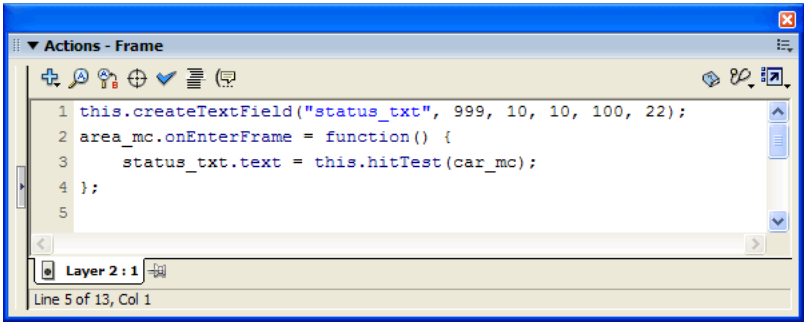
There are two cases in which you would want to know whether a collision has occurred: to test if the user has arrived at a certain static area on the Stage or to determine when one movie clip has reached another. With `hitTest()`, you can determine these results.

You can use the parameters of `hitTest()` to specify the *x* and *y* coordinates of a hit area on the Stage or use the target path of another movie clip as a hit area. When specifying *x* and *y*, `hitTest()` returns `true` if the point identified by (*x*, *y*) is a non-transparent point. When a target is passed to `hitTest()`, the bounding boxes of the two movie clips are compared. If they overlap, `hitTest()` returns `true`. If the two boxes do not intersect, `hitTest()` returns `false`.



"True" appears in the text field whenever the pointer is over the car body.

You can also use `hitTest()` to test a collision between two movie clips.



"True" appears in the text field whenever one movie clip touches the other.

The following procedures show how to detect collision using the car example.

To detect a collision between a movie clip and a point on the Stage:

1. Create a new movie clip on the Stage, and enter `box_mc` as the instance name in the Property inspector.
2. Select the first frame on Layer 1 in the Timeline.
3. Select Window > Development Panels > Actions to open the Actions panel, if it is not already open.
4. Add the following code in the Actions panel:

```
this.createTextField("status_txt", 999, 0, 0, 100, 22);
status_txt.border = true;

box_mc.onEnterFrame = function() {
  status_txt.text = this.hitTest(_xmouse, _ymouse, true);
};
```

5. Select Control > Test Movie, and move the pointer over the movie clip to test the collision.

   The value `true` appears whenever the pointer is over a non-transparent pixel.

To perform collision detection on two movie clips:

1. Drag two movie clips to the Stage, and give them the instance names `car_mc` and `area_mc`.
2. Select the first frame on Layer 1 in the Timeline.
3. Select Window > Development Panels > Actions to open the Actions panel, if it is not already visible.
4. Enter the following code in the Actions panel:

```
this.createTextField("status_txt", 999, 10, 10, 100, 22);
area_mc.onEnterFrame = function() {
  status_txt.text = this.hitTest(car_mc);
};

car_mc.onPress = function() {
  this.startDrag(false);
  updateAfterEvent();
};
car_mc.onRelease = function() {
  this.stopDrag();
};
```

5. Select Control > Test Movie, and drag the movie clip to test the collision detection.

   Whenever the bounding box of the car intersects the bounding box of the area, the status is `true`.

For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help. For more information, see `MovieClip.hitTest()` in ActionScript Dictionary Help.

Creating a simple line drawing tool

You can use methods of the MovieClip class to draw lines and fills on the Stage as the SWF file plays. This lets you create drawing tools for users and draw shapes in the SWF file in response to events. The drawing methods are `beginFill()`, `beginGradientFill()`, `clear()`, `curveTo()`, `endFill()`, `lineTo()`, `lineStyle()`, and `moveTo()`. You can apply these methods to any movie clip instance (for example, `myClip.lineTo()`) or to a level (`_level0.curveTo()`).

The `lineTo()` and `curveTo()` methods let you draw lines and curves, respectively. You specify a line color, thickness, and alpha setting for a line or curve with the `lineStyle()` method. The `moveTo()` drawing method sets the current drawing position to the *x* and *y* Stage coordinates you specify.

The `beginFill()` and `beginGradientFill()` methods fill a closed path with a solid or gradient fill, respectively, and `endFill()` applies the fill specified in the last call to `beginFill()` or `beginGradientFill()`. The `clear()` method erases what's been drawn in the specified movie clip object.

For more information about the methods of the Color class, see the Color class entry in ActionScript Dictionary Help.<<confirm correct book name -bh>>For more information, see `MovieClip.beginFill()`, `MovieClip.beginGradientFill()`, `MovieClip.clear()`, `MovieClip.curveTo()`, `MovieClip.endFill()`, `MovieClip.lineTo()`, `MovieClip.lineStyle()`, and `MovieClip.moveTo()` in ActionScript Language Reference Help.

To create a simple line drawing tool:

1. In a new document, create a button on the Stage, and enter `clear_btn` as the instance name in the Property inspector.
2. Select Frame 1 in the Timeline.
3. Select Window > Development Panels > Actions to open the Actions panel, if it is not already visible.
4. In the Actions panel, enter the following code:

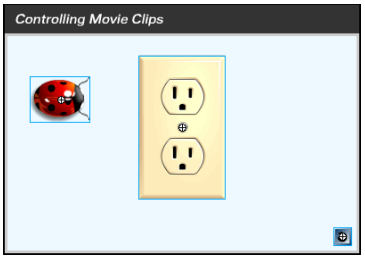
```
this.createEmptyMovieClip("canvas_mc", 999);
var isDrawing:Boolean = false;
//
clear_btn.onRelease = function() {
  canvas_mc.clear();
};
//
var mouseListener:Object = new Object();
mouseListener.onMouseDown = function() {
  canvas_mc.lineStyle(5, 0xFF0000, 100);
  canvas_mc.moveTo(_xmouse, _ymouse);
  isDrawing = true;
};
mouseListener.onMouseMove = function() {
  if (isDrawing) {
    canvas_mc.lineTo(_xmouse, _ymouse);
    updateAfterEvent();
  }
};
```

```
mouseListener.onMouseUp = function() {
  isDrawing = false;
};
Mouse.addListener(mouseListener);
```

5. Select Control > Test Movie to test the movie. Click and drag your pointer to draw a line on the Stage. Click the button to erase what you've drawn.

Deconstructing a sample script

In the sample SWF file zapper.swf (which you can view in Using Flash Help), when a user drags the bug to the electrical outlet, the bug falls and the outlet shakes. The main Timeline has only one frame and contains three objects: the ladybug, the outlet, and a reset button. Each object is a movie clip instance.



The following script is attached to Frame 1 of the main Timeline:

```
var initx:Number = bug_mc._x;
var inity:Number = bug_mc._y;

reset_btn.onRelease = function() {
  zapped = false;
  bug_mc._x = initx;
  bug_mc._y = inity;
  bug_mc._alpha = 100;
  bug_mc._rotation = 0;
};

bug_mc.onPress = function() {
  this.startDrag();
};
bug_mc.onRelease = function() {
  this.stopDrag();
};
bug_mc.onEnterFrame = function() {
  if (this.hitTest(this._parent.zapper_mc)) {
    this.stopDrag();
    zapped = true;
    bug_mc._alpha = 75;
    bug_mc._rotation = 20;
    this._parent.zapper_mc.play();
  }
```

```
  if (zapped) {
    bug_mc._y += 25;
  }
};
```

The bug's instance name is `bug_mc`, and the outlet's instance name is `zapper_mc`. In the script, the bug is referred to as `this` because the script is attached to the bug and the reserved word `this` refers to the object that contains it.

There are event handlers with several different events: `onRelease`, `onPress`, and `enterFrame`. The event handlers are defined on Frame 1 after the SWF file loads. The actions in the `onEnterFrame()` event handler executes every time the playhead enters a frame. Even in a one-frame SWF file, the playhead still enters that frame repeatedly and the script executes repeatedly.

Two variables, `initx` and `inity`, are defined to store the initial *x* and *y* positions of the `bug_mc` movie clip instance. A function is defined and assigned to the `onRelease` event handler of the `reset_btn` instance. This function is called each time the mouse button is pressed and released on the `reset_btn` button. The function places the ladybug back in its starting position on the Stage, resets its rotation and alpha values, and resets the `zapped` variable to `false`.

A conditional `if` statement uses the `hitTest()` method to check whether the bug instance is touching the outlet instance (`this._parent.zapper_mc`). There are two possible outcomes of the evaluation, `true` or `false`:

- If the `hitTest()` method returns `true`, the `stopDrag()` method is called, the `zapper_mc` variable is set to `true`, the alpha and rotation properties are changed, and the `zapped` instance is told to play.
- If the `hitTest()` method returns `false`, none of the code within the curly braces (`{}`) immediately following the `if` statement runs.

The actions in the `onPress()` statement execute when the mouse button is pressed over the `bug_mc` instance. The actions in the `onRelease()` statement execute when the mouse button is released over the `bug_mc` instance.

The `startDrag()` action lets you drag the ladybug. Because the script is attached to the `bug_mc` instance, the keyword `this` indicates that it is the `bug` instance that is draggable:

```
bug_mc.onPress = function() {
  this.startDrag();
};
```

The `stopDrag()` action stops the drag action:

```
bug_mc.onRelease = function() {
  this.stopDrag();
};
```

To watch the SWF file play, see this section in Using ActionScript Help in Flash.

CHAPTER 7
Using the Built-In Classes

In addition to the ActionScript core language elements and constructs (`for` and `while` loops, for example) and primitive data types (numbers, strings, and arrays) described earlier in this manual (see "ActionScript Basics" on page 23), ActionScript also provides several built-in classes (*complex data types*). These classes provide a variety of scripting features and functionality.

Some of these classes are based on the ECMAScript specification and are called *core ActionScript classes.* Examples of core classes are the Array, Boolean, Date, and Math classes. For a complete list, see "Core classes" on page 199.

The rest of the built-in ActionScript classes are specific to Macromedia Flash and the Flash Player object model. Examples of these classes are the Camera, MovieClip, and LoadVars classes. For a complete list, see "Classes specific to Flash Player" on page 200.

To understand the distinction between core ActionScript classes and those specific to Flash, consider the distinction between core and client-side JavaScript: The client-side JavaScript classes provide control over the client environment (the web browser and web page content), and the classes specific to Flash provide runtime control over the appearance and behavior of a Flash application.

This chapter introduces the built-in ActionScript classes, describes common tasks you can perform with these classes, and provides code examples. For an overview of these classes, see "Overview of built-in classes" on page 199. For an overview of working with classes and objects in object-oriented programming, see "About classes and instances" on page 197.

About classes and instances

In object-oriented programming, a *class* defines a category of object. A class describes the properties (data) and behavior (methods) for an object, much like an architectural blueprint describes the characteristics of a building. To use the properties and methods defined by a class, you generally first create an *instance* of that class (except for classes that have static members). The relationship between an instance and its class is similar to the relationship between a house and its architectural blueprints.

For more information, see the following topics:

- "Creating a new object" on page 198
- "Accessing object properties" on page 198
- "Calling object methods" on page 198
- "About class (static) members" on page 199

Creating a new object

To create an instance of an ActionScript class, use the `new` operator to invoke the class's constructor function. The constructor function always has the same name as the class, and returns an instance of the class, which you typically assign to a variable.

For example, the following code creates a new Sound object:

```
var song:Sound= new Sound();
```

In some cases, you don't need to create an instance of a class to use its properties and methods. For more information, see "About class (static) members" on page 199.

Accessing object properties

Use the dot (`.`) operator to access the value of a property in an object. Put the name of the object on the left side of the dot, and put the name of the property on the right side. For example, in the following statement, `myObject` is the object and `name` is the property:

```
myObject.name
```

The following code creates a new Array object and then shows its `length` property:

```
var my_array:Array = new Array("apples", "oranges", "bananas");
trace(my_array.length); // length is 3
```

You can also use the array access operator (`[]`) to access the properties of an object, such as using the array access operator for debugging purposes. The following code loops over an object to display each of its properties:

```
for (i in results) {
    trace("the value of ["+i+"] is: "+ results[i]);
}
```

For more information, see "Dot and array access operators" on page 56.

Calling object methods

You call an object's method by using the dot (`.`) operator followed by the method. For example, the following code creates a new Sound object and calls its `setVolume()` method:

```
var mySound:Sound = new Sound(this);
mySound.setVolume(50);
```

About class (static) members

Some built-in ActionScript classes have *class members* (or *static members*). Class members (properties and methods) are accessed or invoked on the class name, not on an instance of the class. Therefore, you don't create an instance of the class to use those properties and methods.

For example, all the properties of the Math class are static. The following code invokes the `max()` method of the Math class to determine the larger of two numbers:

```
var largerNumber:Number = Math.max(10, 20);
```

Overview of built-in classes

This section lists all the ActionScript classes, including a brief description of each class and cross-references to other relevant sections of the documentation.

For purposes of understanding ActionScript, the classes are divided into the following two categories:

- "Core classes" on page 199
- "Classes specific to Flash Player" on page 200

Core classes

The core ActionScript classes are borrowed directly from ECMAScript. In the Actions toolbox, these classes are located in the Built-in Classes > Core directory.

Class	Description
arguments	An array that contains the values that were passed as parameters to any function. See "arguments object" in *Flash ActionScript Language Reference*
Array	The Array class represents arrays in ActionScript and all array objects are instances of this class. The Array class contains methods and properties for working with array objects. See "Array class" in *Flash ActionScript Language Reference*.
Boolean	The Boolean class is a wrapper for Boolean (`true` or `false`) values. See "Boolean class"in *Flash ActionScript Language Reference*.
Date	The Date class shows how dates and times are represented in ActionScript, and it supports operations for manipulating dates and times. The Date class also provides the means for obtaining the current date and time from the operating system. See "Date class"in *Flash ActionScript Language Reference*.
Error	The Error class contains information about runtime errors that occur in your scripts. You typically use the `throw` statement to generate an error condition, which you can handle using a `try.catch.finally` statement. See `try..catch..finally` and "Error class"in *Flash ActionScript Language Reference*.
Function	The Function class is the class representation of all ActionScript functions, including those native to ActionScript and those that you define. See "Function class" in *Flash ActionScript Language Reference*.

Class	Description
Math	The Math class provides convenient access to common mathematical constants and provides several common mathematical functions. All the properties and methods of the Math class are static and must be called with the syntax `Math.method(parameter)` or `Math.constant`. See "Math class"in *Flash ActionScript Language Reference*.
Number	The Number class is a wrapper for the primitive number data type. See "Number class"in *Flash ActionScript Language Reference*.
Object	The Object class is at the root of the ActionScript class hierarchy; all other classes inherit its methods and properties. See "Object class"in *Flash ActionScript Language Reference*.
String	The String class is a wrapper for the string primitive data type, which lets you use the methods and properties of the String object to manipulate primitive string value types. See "String class"in *Flash ActionScript Language Reference*.
System	The System class provides information about Flash Player and the system on which Flash Player is running (for example, screen resolution and current system language). It also lets you show or hide the Flash Player Settings panel and modify SWF file security settings. See "System class"in *Flash ActionScript Language Reference*.

Classes specific to Flash Player

The following sections list the classes that are specific to Flash Player and the Flash runtime model. For the purposes of understanding ActionScript, these classes are typically grouped into four categories:

- "Media classes" on page 201 for working with sound and video
- "Movie classes" on page 202, which provide overall control over most visual elements in SWF files and over Flash Player
- "Client-server classes" on page 203 for working with XML and other external data sources
- "Authoring classes" on page 204, which provide control for SWF files used in the Flash authoring environment

This categorization affects the locations of the classes in the Actions toolbox but not how you use the classes.

Media classes

The media classes provide playback control of sound and video in a SWF file as well as access to the user's microphone and camera, if they are installed. These classes are located in the Built-in Classes > Media directory in the Actions toolbox.

Class	Description
Camera	The Camera class provides access to the user's camera, if one is installed. When used with Flash Communication Server MX, your SWF file can capture, broadcast, and record images and video from a user's camera. See "Camera class"in *Flash ActionScript Language Reference*.
Microphone	The Microphone class provides access to the user's microphone, if one is installed. When used with Flash Communication Server MX, your SWF file can broadcast and record audio from a user's microphone. See "Microphone class"in *Flash ActionScript Language Reference*.
NetConnection	The NetConnection class is used to establish a local streaming connection for playing a Flash Video (FLV) file from an HTTP address or from the local file system. For more information, see "NetConnection class"in *Flash ActionScript Language Reference*. For more information on playing FLV files over the Internet, see "About playing back external FLV files dynamically" in *Using Flash*.
NetStream	The NetStream class is used to control playback of FLV files. For more information, see "NetStream class"in *Flash ActionScript Language Reference*. For more information on playing FLV files over the Internet, see "About playing back external FLV files dynamically" on page 183.
Sound	The Sound class provides control over sounds in a SWF file. For more information, see "Sound class"in *Flash ActionScript Language Reference*. For an example of using the Sound class to create volume and balance controllers, see "Creating sound controls" on page 189.
Video	The Video class is used to show video objects in a SWF file. See "Video class"in *Flash ActionScript Language Reference*.

Movie classes

The movie classes provide control over most visual elements in a SWF file, including movie clips, text fields, and buttons. The movie classes are located in the Actions toolbox in the Built-in Classes > Movie directory.

Class	Description
Accessibility	The Accessibility class manages communication between SWF files and screen reader applications. You use the methods of this class with the global `_accProps` property to control accessible properties for movie clips, buttons, and text fields at runtime. See "Accessibility class"in *Flash ActionScript Language Reference*.
Button	The Button class provides methods, properties, and event handlers for working with buttons. See "Button class"in *Flash ActionScript Language Reference*. The built-in Button class is different from the Button component class. For information on the Button component class, see "Using the Button component" in *Using Components*.
Color	The Color class lets you get and set RGB color values for movie clip objects. For more information, see "Color class"in *Flash ActionScript Language Reference*. For an example of using the Color class to change the color of movie clips, see "Setting color values" on page 187.
ContextMenu	The ContextMenu class lets you control the contents of the Flash Player context menu. You can associate separate ContextMenu objects with MovieClip, Button, or TextField objects by using the `menu` property available to those classes. You can also add custom menu items to a ContextMenu object by using the ContextMenuItem class. See "ContextMenu class" and "ContextMenuItem class" in *Flash ActionScript Language Reference*.
ContextMenuItem	The ContextMenuItem class lets you create new menu items that appear in the Flash Player context menu. You add new menu items that you create with this class to the Flash Player context menu by using the ContextMenu class. See "ContextMenu class" and "ContextMenuItem class"in *Flash ActionScript Language Reference*.
Key	The Key class provides methods and properties for getting information about the keyboard and keypresses. See "Key class"in *Flash ActionScript Language Reference*. For an example of capturing keypresses to create an interactive SWF file, see "Capturing keypresses" on page 184.
LocalConnection	The LocalConnection class lets two SWF files running on the same computer or web page communicate. This communication can be cross-domain. See "LocalConnection class"in *Flash ActionScript Language Reference*.
Mouse	The Mouse class provides control over the mouse in a SWF file; for example, this class lets you hide or show the mouse pointer. See "Mouse class"in *Flash ActionScript Language Reference*. For an example of using the Mouse class, see "Creating a custom mouse pointer" on page 182.
MovieClip	Every movie clip in a SWF file is an instance of the MovieClip class. You use the methods and properties of this class to control movie clip objects. See the "MovieClip class" in *Flash ActionScript Language Reference* and Chapter 8, "Working with Movie Clips," on page 205.

Class	Description
MovieClipLoader	The MovieClipLoader class lets you track the download progress of SWF and JPEG files using an event listener mechanism. See the "MovieClipLoader class" in *Flash ActionScript Language Reference* and "Preloading SWF and JPEG files" on page 301.
PrintJob	The PrintJob class lets you print content from a SWF file, including content that is rendered dynamically and multi-page documents. See "PrintJob class" in *Flash ActionScript Language Reference* and "Using the ActionScript PrintJob class" in *Using Flash*.
Selection	The Selection class lets you get and set text field focus, text field selection spans, and text field insertion points. See "Selection class" in *Flash ActionScript Language Reference*.
SharedObject	The SharedObject class offers persistent local data storage on the client computer, similar to cookies. See "SharedObject class" in *Flash ActionScript Language Reference*.
Stage	The Stage class provides information about a SWF file's dimensions, alignment, and scale mode. It also reports Stage resize events. See "Stage class" in *Flash ActionScript Language Reference*.
TextField	The TextField class provides control over dynamic and input text fields, such as retrieving formatting information, invoking event handlers, and changing properties such as alpha or background color. See Chapter 9, "Using the TextField class," on page 221 and "TextField class" in *Flash ActionScript Language Reference*.
TextField.StyleSheet	The TextField.StyleSheet class (an inner class of the TextField class) lets you create and apply CSS text styles to HTML- or XML-formatted text. See "Formatting text with Cascading Style Sheets" on page 226 and "TextField.StyleSheet class" in *Flash ActionScript Language Reference*.
TextFormat	The TextFormat class lets you apply formatting styles to characters or paragraphs in a TextField object. See "Using the TextFormat class" on page 224 and "TextFormat class" in *Flash ActionScript Language Reference*.
TextSnapshot	The TextSnapshot object lets you access and lay out static text inside a movie clip. See "TextSnapshot object"in *Flash ActionScript Language Reference*.

Client-server classes

The following table lists classes that let you send and receive data from external sources or communicate with application servers over FTP, HTTP, or HTTPS.

Note: In Flash Player 7, a SWF file can load data only from exactly the same domain from which it was served. For more information, see "New security model and legacy SWF files" on page 13 and "Cross-domain and subdomain access between SWF files" on page 15.

These classes are located in the Built-in Classes > Client/Server folder in the Actions toolbox.

Class	Description
LoadVars	The LoadVars class is an alternative to the `MovieClip.loadVariables()` action for transferring variables between a SWF file and a server in name-value pairs. See "LoadVars class" and "Using the LoadVars class" in *Flash ActionScript Language Reference*.
XML	The XML class extends the XMLNode class and provides methods, properties, and event handlers for working with XML-formatted data, including loading and parsing external XML, creating new XML documents, and navigating XML document trees. See "XML class" and "Using the XML class" in *Flash ActionScript Language Reference*.
XMLNode	The XMLNode class represents a single node in an XML document tree. It is the XML class' superclass. See "XMLNode class" in *Flash ActionScript Language Reference*.
XMLSocket	The XMLSocket class lets you create a persistent socket connection with another computer for low-latency data transfer, such as that required for real-time chat applications. See "Using the XMLSocket class" on page 284 and "XMLSocket class" in *Flash ActionScript Language Reference*.

Authoring classes

The authoring classes are available only in the Flash authoring environment. These classes are found in the Built-in Classes > Authoring directory in the Actions toolbox.

Class	Description
CustomActions	The CustomActions class lets you manage any custom actions that are registered with the authoring tool. See "CustomActions class"in *Flash ActionScript Language Reference*.
Live Preview	The Live Preview feature (listed under Built-in Classes in the Actions toolbox, but not a class) provides a single function called `onUpdate` that is used by component developers. See `onUpdate` in *Flash ActionScript Language Reference*.

CHAPTER 8
Working with Movie Clips

Movie clips are self-contained SWF files that run independently of each other and the Timeline that contains them. For example, if the main Timeline has only one frame and a movie clip in that frame has ten frames, each frame in the movie clip plays when you play the main SWF file. A movie clip can, in turn, contain other movie clips, or *nested clips*. Movie clips nested in this way have a hierarchical relationship, where the *parent clip* contains one or more *child clips*.

You can name movie clip instances to uniquely identify them as objects that can be controlled with ActionScript. When you give a movie clip instance an *instance name*, the instance name identifies it as an object of the MovieClip class type. You use the properties and methods of the MovieClip class to control the appearance and behavior of movie clips at runtime.

You can think of movie clips as autonomous objects that can respond to events, send messages to other movie clip objects, maintain their state, and manage their child clips. In this way, movie clips provide the foundation of *component-based architecture* in Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004. In fact, the components available in the Components panel (Window > Development Panels > Components) are sophisticated movie clips that have been designed and programmed to look and behave in certain ways. For information on creating components, see "Creating Components" in *Using Components.*

About controlling movie clips with ActionScript

You can use global ActionScript functions or the methods of the MovieClip class to perform tasks on movie clips. Some MovieClip methods perform the same tasks as functions of the same name; other MovieClip methods, such as `hitTest()` and `swapDepths()`, don't have corresponding function names.

The following example shows the difference between using a method and using a function. Each statement duplicates the instance `my_mc`, names the new clip `newClip`, and places it at a depth of 5.

```
my_mc.duplicateMovieClip("newClip", 5);
duplicateMovieClip("my_mc", "newClip", 5);
```

When a function and a method offer similar behaviors, you can select to control movie clips by using either one. The choice depends on your preference and your familiarity with writing scripts in ActionScript. Whether you use a function or a method, the target Timeline must be loaded in Flash Player when the function or method is called.

To use a method, invoke it by using the target path of the instance name, a dot (.), and then the method name and parameters, as shown in the following statements:

```
myMovieClip.play();
parentClip.childClip.gotoAndPlay(3);
```

In the first statement, `play()` moves the playhead in the `myMovieClip` instance. In the second statement, `gotoAndPlay()` sends the playhead in `childClip` (which is a child of the instance `parentClip`) to Frame 3 and continues to move the playhead.

Global functions that control a Timeline have a *`target`* parameter that let you specify the target path to the instance that you want to control. For example, in the following script `startDrag()` targets the instance the code is placed on and makes it draggable:

```
my_mc.onPress = function() {
  startDrag(this);
};
my_mc.onRelease = function() {
  stopDrag();
};
```

The following functions target movie clips: `loadMovie()`, `unloadMovie()`, `loadVariables()`, `setProperty()`, `startDrag()`, `duplicateMovieClip()`, and `removeMovieClip()`. To use these functions, you must enter a target path for the function's *`target`* parameter to indicate the target of the function.

The following MovieClip methods can control movie clips or loaded levels and do not have equivalent functions: `MovieClip.attachMovie()`, `MovieClip.createEmptyMovieClip()`, `MovieClip.createTextField()`, `MovieClip.getBounds()`, `MovieClip.getBytesLoaded()`, `MovieClip.getBytesTotal()`, `MovieClip.getDepth()`, `MovieClip.getInstanceAtDepth()`, `MovieClip.getNextHighestDepth()`, `MovieClip.globalToLocal()`, `MovieClip.localToGlobal()`, `MovieClip.hitTest()`, `MovieClip.setMask()`, `MovieClip.swapDepths()`.

For more information about these functions and methods, see their entries in *Flash ActionScript Language Reference*.

Calling multiple methods on a single movie clip

You can use the `with` statement to address a movie clip once and then execute a series of methods on that clip. The `with` statement works on all ActionScript objects (for example, Array, Color, and Sound)—not only movie clips.

The `with` statement takes a movie clip as a parameter. The object you specify is added to the end of the current target path. All actions nested inside a `with` statement are carried out inside the new target path, or scope. For example, in the following script, the `with` statement is passed the object `donut.hole` to change the properties of `hole`:

```
with (donut.hole){
  _alpha = 20;
  _xscale = 150;
  _yscale = 150;
}
```

The script behaves as if the statements inside the `with` statement were called from the Timeline of the `hole` instance. The previous code is equivalent to the following example:

```
donut.hole._alpha = 20;
donut.hole._xscale = 150;
donut.hole._yscale = 150;
```

The previous code is also equivalent to the following example:

```
with (donut){
  hole._alpha = 20;
  hole._xscale = 150;
  hole._yscale = 150;
}
```

Loading and unloading additional SWF files

To play additional SWF files without closing Flash Player, or to switch SWF files without loading another HTML page, you can use one of the following options:

- The global `loadMovie()` function or `loadMovie()` method of the MovieClip class.
- The `loadClip()` method of the MovieClipLoader class. For more information on the MovieClipLoader class, see the `MovieClipLoader class` in *Flash ActionScript Language Reference*.

You can also use `loadMovie()` to send variables to a CGI script, which generates a SWF file as its CGI output. For example, you might use this to load dynamic SWF or JPEG files based on specified variables within a movie clip. When you load a SWF file, you can specify a level or movie clip target into which the SWF file will load. If you load a SWF file into a target, the loaded SWF file inherits the properties of the targeted movie clip. After the movie is loaded, you can change those properties.

The `unloadMovie()` method removes a SWF file previously loaded by `loadMovie()`. Explicitly unloading SWF files with `unloadMovie()` ensures a smooth transition between SWF files and can decrease the memory required by Flash Player. It can be more efficient in some situations to set the movie clip's `_visible` property to `false` instead of unloading the clip. If you might reuse the clip at a later time, set the `_visible` property to `false` and then set to `true` when necessary.

Use `loadMovie()` to do any of the following:

- Play a sequence of banner ads that are SWF files by placing a `loadMovie()` function in a container SWF file that sequentially loads and unloads SWF banner files.

- Develop a branching interface with links that lets the user select among several SWF files that are used to display a site's content.
- Build a navigation interface with navigation controls in level 0 that loads content into other levels. Loading content into levels helps produce smoother transitions between pages of content than loading new HTML pages in a browser.

For more information on loading movies, see "Loading external SWF and JPEG files" on page 296.

Specifying a root Timeline for loaded SWF files

The `_root` ActionScript property specifies or contains a reference to the root Timeline of a SWF file. If a SWF file has multiple levels, the root Timeline is on the level that contains the currently executing script. For example, if a script in level 1 evaluates `_root`, `_level1` is returned. However, the Timeline specified by `_root` can change, depending on whether a SWF file is running independently (in its own level) or has been loaded into a movie clip instance by a `loadMovie()` call.

In the following example, consider a file named container.swf that has a movie clip instance named `target_mc` on its main Timeline. The container.swf file declares a variable named `userName` on its main Timeline; the same script then loads another file called contents.swf into the movie clip `target_mc`.

```
// In container.swf:
_root.userName = "Tim";
target_mc.loadMovie("contents.swf");
myButton.onRelease = function() {
  trace(_root.userName);
};
```

In the following example, the loaded SWF file, contents.swf, also declares a variable named `userName` on its root Timeline:

```
// In contents.swf:
_root.userName = "Mary";
```

After contents.swf loads into the movie clip in container.swf, the value of `userName` that's attached to the root Timeline of the hosting SWF file (container.swf) would be set to `"Mary"` instead of `"Tim"`. This could cause code in container.swf (as well as contents.swf) to malfunction.

To force `_root` to always evaluate to the Timeline of the loaded SWF file, rather than the actual root Timeline, use the `_lockroot` property. This property can be set either by the loading SWF file or the SWF file being loaded. When `_lockroot` is set to `true` on a movie clip instance, that movie clip will act as `_root` for any SWF file loaded into it. When `_lockroot` is set to `true` within a SWF file, that SWF file will act as its own root, no matter what other SWF file loads it. Any movie clip, and any number of movie clips, can set `_lockroot` to `true`. By default, this property is `false`.

For example, the author of container.swf could put the following code on Frame 1 of the main Timeline:

```
// Added to Frame 1 in container.swf:
target_mc._lockroot = true;
```

This step ensures that any references to `_root` in contents.swf—or any SWF file loaded into `target_mc`—will refer to its own Timeline, not the actual root Timeline of container.swf. Now when you click the button, "`Tim`" appears.

Equivalently, the author of contents.swf could add the following code to its main Timeline:

```
// Added to Frame 1 in contents.swf:
this._lockroot = true;
```

This would ensure that no matter where contents.swf is loaded, any reference it makes to `_root` will refer to its own main Timeline, not that of the hosting SWF file.

For more information, see `MovieClip._lockroot`.

Loading JPEG files into movie clips

You can use the `loadMovie()` function, or the MovieClip method of the same name, to load JPEG image files into a movie clip instance. You can also use the `loadMovieNum()` function to load a JPEG file into a level.

When you load an image into a movie clip, the upper left corner of the image is placed at the registration point of the movie clip. Because this registration point is often the center of the movie clip, the loaded image might not appear centered. Also, when you load an image to a root Timeline, the upper left corner of the image is placed on the upper left corner of the Stage. The loaded image inherits rotation and scaling from the movie clip, but the original content of the movie clip is removed.

For more information, see `loadMovie()`, `MovieClip.loadMovie()`, and `loadMovieNum()` in *Flash ActionScript Language Reference* and "Loading external SWF and JPEG files" on page 296.

Changing movie clip position and appearance

To change the properties of a movie clip as it plays, write a statement that assigns a value to a property or use the `setProperty()` function. For example, the following code sets the rotation of instance `mc` to 45:

```
this.my_mc._rotation = 45;
```

This is equivalent to the following code, which uses the `setProperty()` function:

```
setProperty("mc", _rotation, 45);
```

Some properties, called *read-only properties,* have values that you can read but cannot set. (These properties are specified as read-only in their ActionScript Language Reference entries.) The following are read-only properties: `_currentframe`, `_droptarget`, `_framesloaded`, `_parent`, `_target`, `_totalframes`, `_url`, `_xmouse`, and `_ymouse`.

You can write statements to set any property that is not read-only. The following statement sets the `_alpha` property of the movie clip instance `wheel`, which is a child of the `car` instance:

```
car.wheel._alpha = 50;
```

In addition, you can write statements that get the value of a movie clip property. For example, the following statement gets the value of the _xmouse property on the current level's Timeline and sets the _x property of the my_mc instance to that value:

```
this.onEnterFrame = function() {
  my_mc._x = this._xmouse;
};
```

This is equivalent to the following code, which uses the getProperty() function:

```
this.onEnterFrame = function() {
  my_mc._x = getProperty(_root, _xmouse);
};
```

The _x, _y, _rotation, _xscale, _yscale, _height, _width, _alpha, and _visible properties are affected by transformations on the movie clip's parent, and transform the movie clip and any of the clip's children. The _focusrect, _highquality, _quality, and _soundbuftime properties are global; they belong only to the level 0 main Timeline. All other properties belong to each movie clip or loaded level.

For a list of movie clip properties, see "Property summary for the MovieClip class" in *Flash ActionScript Language Reference*.

Dragging movie clips

You can use the global startDrag() function or the MovieClip.startDrag() method to make a movie clip draggable. For example, you can make a draggable movie clip for games, drag-and-drop functions, customizable interfaces, scroll bars, and sliders.

A movie clip remains draggable until explicitly stopped by stopDrag() or until another movie clip is targeted with startDrag(). Only one movie clip at a time can be dragged in a SWF file.

To create more complicated drag-and-drop behavior, you can evaluate the _droptarget property of the movie clip being dragged. For example, you might examine the _droptarget property to see if the movie clip was dragged onto a specific movie clip (such as a "trash can" movie clip) and then trigger another action, as shown in the following example:

```
//Drag a piece of garbage
garbage_mc.onPress = function() {
  this.startDrag(false);
};
//When the garbage is dragged over the trashcan, make it invisible
garbage_mc.onRelease = function() {
  this.stopDrag();
  //convert the slash notation to dot notation using eval
  if (eval(this._droptarget) == trashcan_mc) {
    garbage_mc._visible = false;
  }
};
```

For more information, see startDrag() or MovieClip.startDrag() in *Flash ActionScript Language Reference*.

Creating movie clips at runtime

Not only can you create movie clip instances in the Flash authoring environment, you can also create movie clip instances at runtime in the following ways:

- "Creating an empty movie clip" on page 211
- "Duplicating or removing a movie clip" on page 212
- "Attaching a movie clip symbol to the Stage" on page 212

Each movie clip instance you create at runtime must have an instance name and a depth (stacking, or *z*-order) value. The depth you specify determines how the new clip overlaps with other clips on the same Timeline. It also lets you overwrite movie clips that reside at the same depth. (See "Managing movie clip depths" on page 215.)

Creating an empty movie clip

To create a new, empty movie clip instance on the Stage, use the `createEmptyMovieClip()` method of the MovieClip class. This method creates a movie clip as a child of the clip that calls the method. The registration point for a newly created empty movie clip is the upper left corner.

For example, the following code creates a new child movie clip named `new_mc` at a depth of 10 in the movie clip named `parent_mc`:

```
parent_mc.createEmptyMovieClip("new_mc", 10);
```

The following code creates a new movie clip named `canvas_mc` on the root Timeline of the SWF file in which the script is run, and then invokes `loadMovie()` to load an external JPEG file into itself:

```
this.createEmptyMovieClip("canvas_mc", 10);
canvas_mc.loadMovie("flowers.jpg");
```

As shown in the following example, you can load the image picture.jpg into a movie clip and use the `MovieClip.onPress()` method to make the image act like a button. Loading a JPEG using `loadMovie()` replaces the movie clip with the image but doesn't give you access to movie clip methods. To get access to movie clip methods, you must create an empty parent movie clip and a container child movie clip. Load the image into the container and place the event handler on the parent movie clip.

```
//Creates a parent movie clip to hold the container
this.createEmptyMovieClip("mc_1", 0);

//creates a child movie clip inside of "mc_1"
//this is the movie clip the image will replace
mc_1.createEmptyMovieClip("container_mc",99);

//use moviecliploader to load the image
var my_mcl:MovieClipLoader = new MovieClipLoader();
my_mcl.loadClip("picture.jpg", mc_1.container_mc);

//put event handler on the parent movie clip mc_1
mc_1.onPress = function() {
trace("It works");
};
```

For more information, see `MovieClip.createEmptyMovieClip()` in *Flash ActionScript Language Reference*.

Duplicating or removing a movie clip

To duplicate or remove movie clip instances, use the `duplicateMovieClip()` or `removeMovieClip()` global functions, or the MovieClip class methods of the same name. The `duplicateMovieClip()` method creates a new instance of an existing movie clip instance, assigns it a new instance name, and gives it a depth, or *z*-order. A duplicated movie clip always starts at Frame 1, even if the original movie clip was on another frame when duplicated and is always in front of all previously defined movie clips placed on the Timeline.

To delete a movie clip you created with `duplicateMovieClip()`, use `removeMovieClip()`. Duplicated movie clips are also removed if the parent movie clip is deleted.

For more information, see `duplicateMovieClip()` and `removeMovieClip()` in *Flash ActionScript Language Reference*.

Attaching a movie clip symbol to the Stage

The last way to create movie clip instances at runtime is to use `attachMovie()`. The `attachMovie()` method attaches an instance of a movie clip symbol in the SWF file's library to the Stage. The new clip becomes a child clip of the clip that attached it.

To use ActionScript to attach a movie clip symbol from the library, you must export the symbol for ActionScript and assign it a unique linkage identifier. To do this, you use the Linkage Properties dialog box.

By default, all movie clips that are exported for use with ActionScript load before the first frame of the SWF file that contains them. This can create a delay before the first frame plays. When you assign a linkage identifier to an element, you can also specify whether this content should be added before the first frame. If it isn't added in the first frame, you must include an instance of it in some other frame of the SWF file; if you don't, the element will not be exported to the SWF file.

To assign a linkage identifier to movie clip:

1. Select Window > Library to open the Library panel.
2. Select a movie clip in the Library panel.
3. In the Library panel, select Linkage from the Library panel options menu.

   The Linkage Properties dialog box appears.
4. For Linkage, select Export for ActionScript.
5. For Identifier, enter an ID for the movie clip.

   By default, the identifier is the same as the symbol name.
6. You can optionally assign an ActionScript 2.0 class to the movie clip symbol. This lets the movie clip inherit the methods and properties of a specified class. (See "Assigning a class to a movie clip symbol" on page 218.)

7. If you don't want the movie clip to load before the first frame, deselect the Export in First Frame option.

   If you deselect this option, place an instance of the movie clip on the frame of the Timeline where you want it to be available. For example, if the script you're writing doesn't reference the movie clip until Frame 10, then place an instance of the symbol at or before Frame 10 on the Timeline.
8. Click OK.

After you've assigned a linkage identifier to a movie clip, you can attach an instance of the symbol to the Stage at runtime by using `attachMovie()`.

To attach a movie clip to another movie clip:

1. Assign a linkage identifier to a movie clip library symbol, as described in the previous example.
2. With the Actions panel open (Window > Development Panels > Actions), select a frame in the Timeline.
3. In the Actions panel's Script pane, type the name of the movie clip or level to which you want to attach the new movie clip. For example, to attach the movie clip to the root Timeline, type **this**.
4. In the Actions toolbox (at the left of the Actions panel), click Built-in Classes > Movie > MovieClip > Methods, and select `attachMovie()`.
5. Using the code hints that appear as a guide, enter values for the following parameters:
   - For `idName`, specify the identifier you entered in the Linkage Properties dialog box.
   - For `newName`, enter an instance name for the attached clip so that you can target it.
   - For `depth`, enter the level at which the duplicate movie clip will be attached to the movie clip. Each attached movie clip has its own stacking order, with level 0 as the level of the originating movie clip. Attached movie clips are always on top of the original movie clip, as shown in the following example:

     ```
     this.attachMovie("calif_id", "california_mc", 10);
     ```

For more information, see `MovieClip.attachMovie()` in *Flash ActionScript Language Reference*.

Adding parameters to dynamically created movie clips

When you create or duplicate a movie clip dynamically using `MovieClip.attachMovie()` and `MovieClip.duplicateMovie()`, you can populate the movie clip with parameters from another object. The *`initObject`* parameter of `attachMovie()` and `duplicateMovie()` allows dynamically created movie clips to receive clip parameters.

For more information, see `MovieClip.attachMovie()` and `MovieClip.duplicateMovieClip()` in *Flash ActionScript Language Reference*.

To populate a dynamically created movie clip with parameters from a specified object:

Do one of the following:

- Use the following syntax with `attachMovie()`:
  myMovieClip.attachMovie(*idName*, *newName*, *depth* [, *initObject*])
- Use the following syntax with `duplicateMovie()`:
  myMovieClip.duplicateMovie(*idName*, *newName*, *depth* [, *initObject*])

The *initObject* parameter specifies the name of the object whose parameters you want to use to populate the dynamically created movie clip.

To populate a movie clip with parameters by using `attachMovie()`:

1. In a new Flash document, create a movie clip symbol by selecting Insert > New Symbol. Type `dynamic_mc` in the Symbol Name text box, and select the Movie Clip behavior.
2. Inside the symbol, create a dynamic text field on the Stage with an instance name of name_txt. Make sure this text field is below and to the right of the registration point.
3. Select the first frame of the movie clip's Timeline, and open the Actions panel (Window > Development Panels > Actions).
4. Create a new variable called `name`, and assign its value to the `text` property of `name_txt`, as shown in the following example:

```
var name:String;
name_txt.text = name;
```

5. Select Edit > Edit Document to return to the main Timeline.
6. Select the movie clip symbol in the library, and select Linkage from the Library options menu.

   The Linkage Properties dialog box appears.
7. Select the Export for ActionScript option, and click OK.
8. Select the first frame of the main Timeline, and add the following code to the Actions panel's Script pane:

```
/* Attaches a new clip called Erick and

  moves it to an x and y coordinate of 50 */
this.attachMovie("dynamic_mc", "newClip_mc", 99, {name:"Erick", _x:50,
  _y:50});
```

9. Test the movie (Control > Test Movie). The name you specified in the `attachMovie()` call appears inside the new movie clip's text field.

Managing movie clip depths

Every movie clip has its own *z*-order space that determines how objects overlap within its parent SWF file or movie clip. Every movie clip has an associated depth value, which determines if it will render in front of or behind other movie clips in the same movie clip Timeline. When you create a movie clip at runtime using `MovieClip.attachMovie()`, `MovieClip.duplicateMovieClip()`, or `MovieClip.createEmptyMovieClip()`, you always specify a depth for the new clip as a method parameter. For example, the following code attaches a new movie clip to the Timeline of a movie clip named `container_mc` with a depth value of 10.

```
container_mc.attachMovie("symbolID", "clip1_mc", 10);
```

This example creates a new movie clip with a depth of 10 within the *z*-order space of `container_mc`.

The following code attaches two new movie clips to `container_mc`. The first clip, named `clip1_mc`, will render behind `clip2_mc` because it was assigned a lower depth value.

```
container_mc.attachMovie("symbolID", "clip1_mc", 10);
container_mc.attachMovie("symbolID", "clip2_mc", 15);
```

Depth values for movie clips can range from -16384 to 1048575. If you create or attach a new movie clip on a depth that already has a movie clip, the new or attached clip will overwrite the existing content.

The MovieClip class provides several methods for managing movie clip depths; for more information, see `MovieClip.getNextHighestDepth()`, `MovieClip.getInstanceAtDepth()`, `MovieClip.getDepth()`, and `MovieClip.swapDepths()` *Flash ActionScript Language Reference*.

For more information on movie clip depths, see the following topics:

- "Determining the next highest available depth" on page 215
- "Determining the instance at a particular depth" on page 216
- "Determining the depth of an instance" on page 216
- "Swapping movie clip depths" on page 216

Determining the next highest available depth

To determine the next highest available depth within a movie clip, use `MovieClip.getNextHighestDepth()`. The integer value returned by this method indicates the next available depth that will render in front of all other objects in the movie clip.

The following code attaches a new movie clip, with a depth value of 10, on the root Timeline named `file_menu`. It then determines the next highest available depth in that same movie clip and creates a new movie clip called `edit_mc` at that depth.

```
this.attachMovie("menuClip","file_mc", 10, {_x:0, _y:0});
trace(file_menu.getDepth());
var nextDepth = this.getNextHighestDepth();
this.attachMovie("menuClip", "edit_mc", nextDepth, {_x:200, _y:0});
trace(edit_menu.getDepth());
```

In this case, the variable named `nextDepth` contains the value 11 because that's the next highest available depth for the movie clip `edit_mc`.

To obtain the current highest occupied depth, subtract 1 from the value returned by `getNextHighestDepth()`, as shown in the next section.

Determining the instance at a particular depth

To determine the instance at a particular depth, use `MovieClip.getInstanceAtDepth()`. This method returns a reference to the MovieClip instance at the specified depth.

The following code combines `getNextHighestDepth()` and `getInstanceAtDepth()` to determine the movie clip at the (current) highest occupied depth on the root Timeline.

```
var highestOccupiedDepth = this.getNextHighestDepth() - 1;
var instanceAtHighestDepth = this.getInstanceAtDepth(highestOccupiedDepth);
```

For more information, see `MovieClip.getInstanceAtDepth()` in *Flash ActionScript Language Reference*.

Determining the depth of an instance

To determine the depth of a movie clip instance, use `MovieClip.getDepth()`.

The following code iterates over all the movie clips on a SWF file's main Timeline and shows each clip's instance name and depth value in the Output panel:

```
for(each in _root) {
  var obj = _root[each];
  if(obj instanceof MovieClip) {
    var objDepth = obj.getDepth();
    trace(obj._name + ":" + objDepth)
  }
}
```

For more information, see `MovieClip.getDepth()` in *Flash ActionScript Language Reference*.

Swapping movie clip depths

To swap the depths of two movie clips on the same Timeline, use `MovieClip.swapDepths()`. For more information, see `MovieClip.swapDepths()` in *Flash ActionScript Language Reference*.

Drawing shapes with ActionScript

You can use methods of the MovieClip class to draw lines and fills on the Stage. This lets you create drawing tools for users and to draw shapes in the movie in response to events. The drawing methods are `beginFill()`, `beginGradientFill()`, `clear()`, `curveTo()`, `endFill()`, `lineTo()`, `lineStyle()`, and `moveTo()`.

You can use the drawing methods with any movie clip. However, if you use the drawing methods with a movie clip that was created in authoring mode, the drawing methods execute before the clip is drawn. In other words, content that is created in authoring mode is drawn on top of content drawn with the drawing methods.

You can use movie clips with drawing methods as masks; however, as with all movie clip masks, strokes are ignored.

To draw a shape:

1. Use `MovieClip.createEmptyMovieClip()` to create an empty movie clip on the Stage.

   The new movie clip is a child of an existing movie clip or of the main Timeline, as shown in the following example:

   ```
   this.createEmptyMovieClip ("triangle_mc", 1);
   ```

2. Use the empty movie clip to call drawing methods.

   The following example draws a triangle with 5-point magenta lines and no fill:

   ```
   with (triangle_mc) {
        lineStyle (5, 0xff00ff, 100);
        moveTo (200, 200);
        lineTo (300, 300);
        lineTo (100, 300);
        lineTo (200, 200);
   }
   ```

For detailed information on these methods, see their entries in the "MovieClip class" in *Flash ActionScript Language Reference*. For a sample that uses the drawing methods of the MovieClip class, see the drawingapi.fla file in the Samples\HelpExamples directory.

Using movie clips as masks

You can use a movie clip as a mask to create a hole through which the contents of another movie clip are visible. The mask movie clip plays all the frames in its Timeline, the same as a regular movie clip. You can make the mask movie clip draggable, animate it along a motion guide, use separate shapes within a single mask, or resize a mask dynamically. You can also use ActionScript to turn a mask on and off.

You cannot use a mask to mask another mask. You cannot set the `_alpha` property of a mask movie clip. Only fills are used in a movie clip that is used as a mask; strokes are ignored.

To create a mask:

1. On the Stage, select a movie clip to be masked.
2. In the Property inspector, enter an instance name for the movie clip, such as `image_mc`.
3. Create a movie clip to be a mask. Give it an instance name in the Property inspector, such as `mask_mc`.

   The masked movie clip will be revealed under all opaque (nontransparent) areas of the movie clip acting as the mask.
4. Select Frame 1 in the Timeline.
5. Open the Actions panel (Window > Development Panels > Actions) if it isn't already open.
6. In the Actions panel, enter the following code:

   ```
   image_mc.setMask(mask_mc);
   ```

For detailed information, see `MovieClip.setMask()` in *Flash ActionScript Language Reference*.

About masking device fonts

You can use a movie clip to mask text that is set in a device font. In order for a movie clip mask on a device font to work properly, the user must have Flash Player 6 (6.0.40.0) or later.

When you use a movie clip to mask text set in a device font, the rectangular bounding box of the mask is used as the masking shape. That is, if you create a nonrectangular movie clip mask for device font text in the Flash authoring environment, the mask that appears in the SWF file will be the shape of the rectangular bounding box of the mask, not the shape of the mask itself.

You can mask device fonts only by using a movie clip as a mask. You cannot mask device fonts by using a mask layer on the Stage.

Handling movie clip events

Movie clips can respond to user events, such as mouse clicks and keypresses, as well as system-level events, such as the initial loading of a movie clip on the Stage. ActionScript provides two ways to handle movie clip events: through event handler methods and through `onClipEvent()` and `on()` event handlers. For more information, see Chapter 5, "Handling Events," on page 167.

Assigning a class to a movie clip symbol

Using ActionScript 2.0, you can create a class that extends the behavior of the built-in MovieClip class and then assign that class to a movie clip library symbol using the Linkage Properties dialog box. Whenever you create an instance of the movie clip to which the class is assigned, it assumes the properties and behaviors defined by the class assigned to it. (For more information about ActionScript 2.0, see Chapter 10, "Creating Custom Classes with ActionScript 2.0," on page 247.)

In a subclass of the MovieClip class, you can provide method definitions for the built-in MovieClip methods and event handlers, such as `onEnterFrame` and `onRelease`. In the following procedure, you'll create a class called MoveRight that extends the MovieClip class; MoveRight defines an `onPress` handler that moves the clip 20 pixels to the right whenever the user clicks the movie clip. In the second procedure, you'll create a movie clip symbol in a new Flash (FLA) document and assign the MoveRight class to that symbol.

To create a movie clip subclass:

1. Create a new directory called BallTest.
2. Create a new ActionScript file by doing one of the following:
   - (Flash MX Professional 2004) Select File > New, and select ActionScript file from the list of document types.
   - (Flash MX 2004) Create a text file in your preferred text editor.
3. Enter the following code in your script:

```
// MoveRight class -- moves clip to the right 20 pixels when clicked
class MoveRight extends MovieClip {
  function onPress() {
    this._x += 20;
```

```
    }
}
```

4. Save the document as MoveRight.as in the BallTest directory.

To assign the class to a movie clip symbol:

1. In Flash, select File > New, select Flash Document from the list of file types, and click OK.
2. Using the Oval tool, draw a circle on the Stage.
3. Select the circle, and select Modify > Convert to Symbol.
4. In the Convert to Symbol dialog box, select Movie Clip as the symbol's behavior, and enter `ball_mc` in the Name text box.
5. Select Advanced to show the options for Linkage, if they aren't already showing.
6. Select the Export for ActionScript option, and type `MoveRight` in the AS 2.0 Class text box. Click OK.
7. Save the file as ball.fla in the BallTest directory (the same directory that contains the MoveRight.as file).
8. Test the movie (Control > Test Movie).

   Each time you click the ball movie clip, it moves 20 pixels to the right.

If you create component properties for a class and want a movie clip to inherit those component properties, you need to take an additional step: with the movie clip symbol selected in the Library panel, select Component Definition from the Library options menu and enter the new class name in the AS 2.0 Class box. For more information, see "Creating Components" in *Using Components*.

Initializing class properties

In the example presented earlier, you added the instance of the Ball symbol to the Stage manually—that is, while authoring. As discussed in "Adding parameters to dynamically created movie clips" on page 213, you can assign parameters to clips you create at runtime using the *`initObject`* parameter of `attachMovie()` and `duplicateMovie()`. You can use this feature to initialize properties of the class you're assigning to a movie clip.

For example, the following class named MoveRightDistance is a variation of the MoveRight class (see "Assigning a class to a movie clip symbol" on page 218). The difference is a new property named `distance`, whose value determines how many pixels a movie clip moves each time it is clicked.

```
// MoveRightDistance class -- moves clip to the right 5 pixels every frame
class MoveRightDistance extends MovieClip {
  // distance property determines how many
  // pixels to move clip each mouse press
  var distance:Number;
  function onPress() {
    this._x += this.distance;
  }
}
```

Assuming this class is assigned to a symbol with a linkage identifier of Ball, the following code creates two new instances of the symbol on the root Timeline of the SWF file. The first instance, named `ball50_mc`, moves 50 pixels each time it is clicked; the second, named `ball125_mc`, moves 125 pixels each time its clicked.

```
this.attachMovie("Ball", "ball50_mc", 10, {distance:50});
this.attachMovie("Ball", "ball125_mc", 20, {distance:125});
```

CHAPTER 9
Working with Text

A dynamic or input text field is a TextField object (an instance of the TextField class). When you create a text field, you can assign it an instance name in the Property inspector. You can use the instance name in ActionScript statements to set, change, and format the text field and its content using the TextField and TextFormat classes.

The methods of the TextField class let you set, select, and manipulate text in a dynamic or input text field that you create during authoring or at runtime. For more information, see "Using the TextField class" on page 221. For information on debugging text fields at runtime, see "Displaying text field properties for debugging" on page 164.

ActionScript also provides several ways to format your text at runtime. The TextFormat class lets you set character and paragraph formatting for TextField objects (see "Using the TextFormat class" on page 224). Flash Player also supports a subset of HTML tags that you can use to format text (see "Using HTML-formatted text" on page 236). Flash Player 7 and later supports the `<img>` HTML tag, which lets you embed not just external images but also external SWF files as well as movie clips that reside in the library (see "Image tag (<img>)" on page 239).

In Flash Player 7 and later, you can apply Cascading Style Sheets (CSS) styles to text fields using the TextField.StyleSheet class. You can use CSS to style built-in HTML tags, define new formatting tags, or apply styles. For more information on using CSS, see "Formatting text with Cascading Style Sheets" on page 226.

You can also assign HTML formatted text, which might optionally use CSS styles, directly to a text field. In Flash Player 7 and later, HTML text that you assign to a text field can contain embedded media (movie clips, SWF files, and JPEG files). The text wraps around the embedded media in the same way that a web browser wraps text around media embedded in an HTML document. For more information, see "Image tag (<img>)" on page 239.

Using the TextField class

The TextField class represents any dynamic or selectable (editable) text field you create using the Text tool in Flash. You use the methods and properties of this class to control text fields at runtime. TextField objects support the same properties as MovieClip objects, with the exception of the `_currentframe`, `_droptarget`, `_framesloaded`, and `_totalframes` properties. You can get and set properties and invoke methods for text fields dynamically.

To control a dynamic or input text field using ActionScript, you must assign it an instance name in the Property inspector. You can then reference the text field with the instance name, and use the methods and properties of the TextField class to control the contents or basic appearance of the text field.

You can also create TextField objects at runtime, and assign them instance names, using the `MovieClip.createTextField()` method. For more information, see "Creating text fields at runtime" on page 223.

For more information on using the TextField class, see the following topics:

- "Assigning text to a text field at runtime" on page 222
- "About text field instance and variable names" on page 222

Assigning text to a text field at runtime

To assign text to a text field, use the `TextField.text` property.

To assign text to a text field at runtime:

1. Using the Text tool, create a text field on the Stage.
2. With the text field selected, in the Property inspector (Window > Properties), select Input Text from the Text Type pop-up menu, and enter `headline_txt` in the Instance Name text box.

   Instance names can consist only of letters, numbers, underscores (_), and dollar signs ($).
3. In the Timeline, select the first frame in Layer 1, and open the Actions panel (Window > Development Panels > Actions).
4. Type the following code in the Actions panel:

   ```
   headline_txt.text = "Brazil wins World Cup";
   ```
5. Select Control > Test Movie to test the movie.

About text field instance and variable names

In the Instance Name text box in the Property inspector, you must assign an instance name to a text field to invoke methods and get and set properties on that text field.

In the Var text box in the Property inspector, you can assign a variable name to a dynamic or input text field. You can then assign values to the variable.

Do not confuse a text field's instance name with its variable name, however. A text field's variable name is simply a variable reference to the text contained by that text field; it is not a reference to an object.

For example, if you assigned a text field the variable name `mytextVar`, you can set the contents of the text field using the following code:

```
var mytextVar:String = "This is what will appear in the text field";
```

However, you can't use the variable name `mytextVar` to set the text field's `text` property. You have to use the instance name, as shown in the following code:

```
//This won't work
mytextVar.text = "A text field variable is not an object reference";

//For input text field with instance name "myField", this will work
myField.text = "This sets the text property of the myField object";
```

In general, use the `TextField.text` property to control the contents of a text field, unless you're targeting a version of Flash Player that doesn't support the TextField class. This will reduce the chances of a variable name conflict, which could result in unexpected behavior at runtime.

Creating text fields at runtime

You can use the `createTextField()` method of the MovieClip class to create an empty text field on the Stage at runtime. The new text field is attached to the Timeline of the movie clip that calls the method. The `createTextField()` method uses the following syntax:

```
movieClip.createTextField(instanceName, depth, x, y, width, height)
```

For example, the following code creates a 30 x 100-pixel text field named `test_txt` with a location of (0,0) and a depth (*z*-order) of 10:

```
this.createTextField("test_txt", 10, 0, 0, 300, 100);
```

You use the instance name specified in the `createTextField()` call to access the methods and properties of the TextField class. For example, the following code creates a new text field named `test_txt`, and modifies its properties to make it a multiline, word-wrapping text field that expands to fit inserted text. Then, it assigns some text to the text field's `text` property.

```
this.createTextField("test_txt", 10, 0, 0, 100, 50);
test_txt.multiline = true;
test_txt.wordWrap = true;
test_txt.autoSize = true;
test_txt.text = "Create new text fields with the MovieClip.createTextField
  method.";
```

You can use the `TextField.removeTextField()` method to remove a text field created with `createTextField()`. The `removeTextField()` method does not work on a text field placed by the Timeline during authoring.

For more information, see `MovieClip.createTextField()` and `TextField.removeTextField()` in *Flash ActionScript Language Reference*.

Note: Some TextField properties, such as `_rotation`, are not available when you create text fields at runtime. You can rotate only embedded fonts.

Using the TextFormat class

You can use the ActionScript TextFormat class to set formatting properties of a text field. The TextFormat class incorporates character and paragraph formatting information. Character formatting information describes the appearance of individual characters: font name, point size, color, and an associated URL. Paragraph formatting information describes the appearance of a paragraph: left margin, right margin, indentation of the first line, and left, right, or center alignment.

To use the TextFormat class, you first create a TextFormat object and set its character and paragraph formatting styles. You then apply the TextFormat object to a text field using the `TextField.setTextFormat()` or `TextField.setNewTextFormat()` methods.

The `setTextFormat()` method changes the text format that is applied to individual characters, to groups of characters, or to the entire body of text in a text field. Newly inserted text, however—such as text entered by a user or inserted with ActionScript—does not assume the formatting specified by a `setTextFormat()` call. To specify the default formatting for newly inserted text, use `TextField.setNewTextFormat()`. For more information, see `TextField.setTextFormat()` and `TextField.setNewTextFormat()` in *Flash ActionScript Language Reference.*

To format a text field with the TextFormat class:

1. In a new Macromedia Flash document, create a text field on the Stage using the Text tool. Type some text in the text field on the Stage, such as "`Bold, italic, 24 point text`".
2. In the Property inspector, type `myText_txt` in the Instance Name text box, select Dynamic from the Text Type pop-up menu, and select Multiline from the Line Type pop-up menu.
3. In the Timeline, select the first frame in Layer 1, and open the Actions panel (Window > Development Panels > Actions).
4. Enter the following code in the Actions panel to create a TextFormat object, set the `bold` and `italic` properties to `true`, and the `size` property to 24:

```
// Create a TextFormat object
var txtfmt:TextFormat = new TextFormat();
// Specify paragraph and character formatting
txtfmt.bold = true;
txtfmt.italic = true;
txtfmt.size = 24;
```

5. Apply the TextFormat object to the text field you created in step 1 using `TextField.setTextFormat()`:

   ```
   myText_txt.setTextFormat(txtfmt);
   ```

   This version of `setTextFormat()` applies the specified formatting to the entire text field. There are two other versions of this method that let you apply formatting to individual characters or groups of characters. For example, the following code applies bold, italic, 24-point formatting to the first three characters you entered in the text field:

   ```
   myText_txt.setTextFormat(0, 3, txtfmt);
   ```

   For more information, see `TextField.setTextFormat()` in *Flash ActionScript Language Reference*.

6. Select Control > Test Movie to test the application.

For more information on using the TextFormat class, see the following topics:

- "Default properties of new text fields" on page 225
- "Getting text metric information" on page 226

Default properties of new text fields

Text fields created at runtime with `createTextField()` receive a default TextFormat object with the following properties:

```
font = "Times New Roman"
size = 12
textColor = 0x000000
bold = false
italic = false
underline = false
url = ""
target = ""
align = "left"
leftMargin = 0
rightMargin = 0
indent = 0
leading = 0
bullet = false
tabStops = [] (empty array)
```

Note: The default font property on the Mac OS X is Times.

For a complete list of TextFormat methods and their descriptions, see "TextFormat class" in *Flash ActionScript Language Reference*.

Getting text metric information

You can use the `TextFormat.getTextExtent()` method to obtain detailed text measurements for a text string that has specific formatting. For example, suppose you need to create, at runtime, a new TextField object containing an arbitrary amount of text that is formatted with a 24-point, bold, Arial font, and a 5-pixel indent. You need to determine how wide or high the new TextField object must be to display all the text. The `getTextExtent()` method provides measurements such as ascent, descent, width, and height.

For more information, see `TextFormat.getTextExtent()` in *Flash ActionScript Language Reference.*

Formatting text with Cascading Style Sheets

Cascading Style Sheets (CSS) are a way to text styles that can be applied to HTML or XML documents. A style sheet is a collection of formatting rules that specify how to format HTML or XML elements. Each rule associates a style name, or *selector*, with one or more style properties and their values. For example, the following style defines a selector named `bodyText`:

```
.bodyText { text-align: left}
```

You can create styles that redefine built-in HTML formatting tags used by Flash Player (such as `<p>` and `<li>`), create style *classes* that can be applied to specific HTML elements using the `<p>` or `<span>` tag's `class` attribute or define new tags.

You use the TextField.StyleSheet class to work with text style sheets. Although the TextField class can be used with Flash Player 6, the TextField.StyleSheet class requires that SWF files target Flash Player 7 or later. You can load styles from an external CSS file or create them natively using ActionScript. To apply a style sheet to a text field that contains HTML- or XML-formatted text, you use the `TextField.styleSheet` property. The styles defined in the style sheet are mapped automatically to the tags defined in the HTML or XML document.

Using styles sheets involves the following three basic steps:

- Create a style sheet object from the TextField.StyleSheet class (for more information see "TextField.StyleSheet class" in *Flash ActionScript Language Reference*).
- Add styles to the style sheet object, either by loading them from an external CSS file or by creating new styles with ActionScript.
- Assign the style sheet to a TextField object that contains XML- or HTML-formatted text.

For more information, see the following topics:

- "Supported CSS properties" on page 227
- "Creating a style sheet object" on page 228
- "Loading external CSS files" on page 228
- "Creating new styles with ActionScript" on page 229
- "Applying styles to a TextField object" on page 230
- "Applying a style sheet to a TextArea component" on page 230
- "Combining styles" on page 231

Supported CSS properties

Flash Player supports a subset of properties in the original CSS1 specification (www.w3.org/TR/REC-CSS1). The following table shows the supported CSS properties and values as well as their corresponding ActionScript property names. (Each ActionScript property name is derived from the corresponding CSS property name; the hyphen is omitted and the subsequent character is capitalized.)

CSS property	ActionScript property	Usage and supported values
text-align	textAlign	Recognized values are `left`, `center`, and `right`.
font-size	fontSize	Only the numeric part of the value is used. Units (px, pt) are not parsed; pixels and points are equivalent.
text-decoration	textDecoration	Recognized values are `none` and `underline`.
margin-left	marginLeft	Only the numeric part of the value is used. Units (px, pt) are not parsed; pixels and points are equivalent.
margin-right	marginRight	Only the numeric part of the value is used. Units (px, pt) are not parsed; pixels and points are equivalent.
font-weight	fontWeight	Recognized values are `normal` and `bold`.
font-style	fontStyle	Recognized values are `normal` and `italic`.
text-indent	textIndent	Only the numeric part of the value is used. Units (px, pt) are not parsed; pixels and points are equivalent.
font-family	fontFamily	A comma-separated list of fonts to use, in descending order of desirability. Any font family name can be used. If you specify a generic font name, it will be converted to an appropriate device font. The following font conversions are available: `mono` is converted to `_typewriter`, `sans-serif` is converted to `_sans`, and `serif` is converted to `_serif`.
color	color	Only hexadecimal color values are supported. Named colors (such as `blue`) are not supported. Colors are written in the following format: `#FF0000`.
display	display	Supported values are `inline`, `block`, and `none`.

Creating a style sheet object

CSS are represented in ActionScript by the TextField.StyleSheet class. This class is available only for SWF files that target Flash Player 7 or later. To create a style sheet object, call the TextField.StyleSheet class's constructor function:

```
var newStyle = new TextField.StyleSheet();
```

To add styles to a style sheet object, you can either load an external CSS file into the object or define the styles in ActionScript. See "Loading external CSS files" on page 228 and "Creating new styles with ActionScript" on page 229.

Loading external CSS files

You can define styles in an external CSS file and then load that file into a style sheet object. The styles defined in the CSS file are added to the style sheet object. To load an external CSS file, you use the `load()` method of the TextField.StyleSheet class. To determine when the CSS file has finished loading, use the style sheet object's `onLoad` event handler.

In the following example, you create and load an external CSS file and use the `TextField.StyleSheet.getStyleNames()` method to retrieve the names of the loaded styles.

To load an external style sheet:

1. In your preferred text or CSS editor, create a new file.
2. Add the following style definitions to the file:

```
/* Filename: styles.css */
.bodyText {
  font-family: Arial,Helvetica,sans-serif;
  font-size: 12px;
}

.headline {
  font-family: Arial,Helvetica,sans-serif;
  font-size: 24px;
}
```

3. Save the CSS file as styles.css.
4. In Flash, create a FLA document.
5. In the Timeline (Window > Timeline), select Layer 1.
6. Open the Actions panel (Window > Development Panels > Actions).

7. Add the following code to the Actions panel:

```
var styles = new TextField.StyleSheet();
styles.load("styles.css");
styles.onLoad = function(ok) {
  if (ok) {
    // display style names
    trace(this.getStyleNames());
  } else {
    trace("Error loading CSS file.");
  }
};
```

Note: In the previous code snippet, `this` in `this.getStyleNames()` refers to the `styles` object you constructed in the first line of ActionScript.

8. Save the FLA file to the same directory that contains styles.css.
9. Test the movie (Control > Test Movie).

   You should see the names of the two styles in the Output panel:

   ```
   .bodyText,.headline
   ```

   If you see "Error loading CSS file." in the Output panel, make sure the FLA file and the CSS file are in the same directory and that you typed the name of the CSS file correctly.

As with all other ActionScript methods that load data over the network, the CSS file must reside in the same domain as the SWF file that is loading the file. (See "Cross-domain and subdomain access between SWF files" on page 15.) For more information on using CSS with Flash, see "TextField.StyleSheet class" in *Flash ActionScript Language Reference*.

Creating new styles with ActionScript

You can create new text styles with ActionScript using the `setStyle()` method of the TextField.StyleSheet class. This method takes two parameters: the name of the style and an object that defines that style's properties.

For example, the following code creates a style sheet object named `styles` that defines two styles that are identical to the ones you already imported (see "Loading external CSS files" on page 228):

```
var styles = new TextField.StyleSheet();
styles.setStyle("bodyText",
  {fontFamily: 'Arial,Helvetica,sans-serif',
  fontSize: '12px'}
);
styles.setStyle("headline",
  {fontFamily: 'Arial,Helvetica,sans-serif',
  fontSize: '24px'}
);
```

Applying styles to a TextField object

To apply a style sheet object to a TextField object, you assign the style sheet object to the text field's `styleSheet` property.

```
textObj_txt.styleSheet = styleSheetObj;
```

Note: Do not confuse the `TextField.styleSheet` *property* with the TextField.StyleSheet *class*. The capitalization indicates the difference.

When you assign a style sheet object to a TextField object, the following changes occur to the text field's normal behavior:

- The text field's `text` and `htmlText` properties, and any variable associated with the text field, always contain the same value and behave identically.
- The text field becomes read-only and cannot be edited by the user.
- The `setTextFormat()` and `replaceSel()` methods of the TextField class no longer function with the text field. The only way to change the field is by altering the text field's `text` or `htmlText` properties or by changing the text field's associated variable.
- Any text assigned to the text field's `text` property, `htmlText` property, or associated variable is stored verbatim; anything written to one of these properties can be retrieved in the text's original form.

Applying a style sheet to a TextArea component

To apply a style sheet to a TextArea component, you create a style sheet object and assign it HTML styles using the TextField.StyleSheet class. You then assign the style sheet to the TextArea component's `styleSheet` property.

The following examples create a style sheet object, `styles`, and assigns it to the `myTextArea` component instance:

```
//Creates a new style sheet object and sets styles for it
var styles = new TextField.StyleSheet();
styles.setStyle("html",
  {fontFamily: 'Arial,Helvetica,sans-serif',
   fontSize: '12px',
   color: '#0000FF'}
);
styles.setStyle("body",
  {color: '#00CCFF',
   textDecoration: 'underline'}
);

styles.setStyle("td",
  {fontFamily: 'Arial,Helvetica,sans-serif',
   fontSize: '24px',
   color: '#006600'}
);

//Assign the style sheet object to myTextArea component
//Set html property to true, set styleSheet property to the style sheet object
myTextArea.styleSheet = styles;
```

```
myTextArea.html = true;

var myVars:LoadVars = new LoadVars();
//var styles = new TextField.StyleSheet();

// Load text to display and define onLoad handler
myVars.load("myText.htm");
myVars.onData = function(content) {
  _root.myTextArea.text = content;
};
```

Combining styles

CSS styles in Flash Player are additive; that is, when styles are nested, each level of nesting can contribute style information, which is added together to result in the final formatting.

The following example shows some XML data assigned to a text field:

```
<sectionHeading>This is a section</sectionHeading>
<mainBody>This is some main body text, with one
<emphasized>emphatic</emphasized> word.</mainBody>
```

For the word *emphatic* in the above text, the `emphasized` style is nested within the `mainBody` style. The `mainBody` style contributes color, font-size, and decoration rules. The `emphasized` style adds a font-weight rule to these rules. The word *emphatic* will be formatted using a combination of the rules specified by `mainBody` and `emphasized`.

Using style classes

You can create style "classes" (not true ActionScript 2.0 classes) that you can apply to a `<p>` or `<span>` tag using either tag's `class` attribute. When applied to a `<p>` tag, the style affects the entire paragraph. You can also style a span of text that uses a style class using the `<span>` tag.

For example, the following style sheet defines two styles classes: `mainBody` and `emphasis`:

```
.mainBody {
  font-family: Arial,Helvetica,sans-serif;
  font-size: 24px;
}
.emphasis {
  color: #666666;
  font-style: italic;
}
```

Within HTML text you assign to a text field, you can apply these styles to `<p>` and `<span>` tags, as shown in the following example:

```
<p class='mainBody'>This is <span class='emphasis'>really exciting!</span></p>
```

Styling built-in HTML tags

Flash Player supports a subset of HTML tags. (For more information, see "Using HTML-formatted text" on page 236.) You can assign a CSS style to every instance of a built-in HTML tag that appears in a text field. For example, the following code defines a style for the built-in `<p>` HTML tag. All instances of that tag are styled in the manner specified by the style rule.

```
p {
  font-family: Arial,Helvetica,sans-serif;
  font-size: 12px;
  display: inline;
}
```

The following table shows which built-in HTML tags can be styled and how each style is applied:

Style name	How the style is applied
`p`	Affects all `<p>` tags.
`body`	Affects all `<body>` tags. The `p` style, if specified, takes precedence over the `body` style.
`li`	Affects all `<li>` bullet tags.
`a`	Affects all `<a>` anchor tags.
`a:link`	Affects all `<a>` anchor tags. This style is applied after any `a` style.
`a:hover`	Applied to an `<a>` anchor tag when the mouse is over the link. This style is applied after any `a` and `a:link` style. After the mouse moves off the link, the `a:hover` style is removed from the link.
`a:active`	Applied to an `<a>` anchor tag when the user clicks the link. This style is applied after any `a` and `a:link` style. After the mouse button is released, the `a:active` style is removed from the link.

An example of using styles with HTML

This section presents an example of using styles with HTML tags. You can create a style sheet that styles some built-in tags and defines some style classes. Then, you can apply that style sheet to a TextField object that contains HTML-formatted text.

To format HTML with a style sheet:

1. In your preferred text or XML editor, create a file.
2. Add the following style sheet definition to the file:

```
p {
  color: #000000;
  font-family: Arial,Helvetica,sans-serif;
  font-size: 12px;
  display: inline;
}

a:link {
  color: #FF0000;
}

a:hover{
  text-decoration: underline;
}

.headline {
  color: #000000;
```

```
  font-family: Arial,Helvetica,sans-serif;
  font-size: 18px;
  font-weight: bold;
  display: block;
}

.byline {
  color: #666600;
  font-style: italic;
  font-weight: bold;
  display: inline;
}
```

This style sheet defines styles for two built-in HTML tags (`<p>` and `<a>`) that will be applied to all instances of those tags. It also defines two style classes (`.headline` and `.byline`) that will be applied to specific paragraphs and text spans.

3. Save the file as html_styles.css.
4. Create a new text file in a text or XML editor, and save the document as myText.txt. Add the following text to the file:

```
<p class='headline'>Flash Player now supports Cascading StyleSheets!</
  p><p><span class='byline'>San Francisco, CA</span>--Macromedia Inc.
  announced today a new version of Flash Player that supports Cascading
  Style Sheet (CSS) text styles. For more information, visit the <a
  href='http://www.macromedia.com'>Macromedia Flash web site.</a></p>
```

Note: If you copy and paste this text string, make sure that you remove any line breaks that might have been added to the text file.

5. In Flash, create a FLA file.
6. Using the Text tool, create a text field approximately 400 pixels wide and 300 pixels high.
7. Open the Property inspector (Window > Properties), and select the text field.
8. In the Property inspector, select Dynamic Text from the Text Type menu, select Multiline from the Line Type menu, select the Render Text as HTML option, and type `news_txt` in the Instance Name text box.
9. Select the first frame in Layer 1 in the Timeline (Window > Timeline).
10. Open the Actions panel (Window > Development Panels > Actions), and add the following code to the Actions panel:

```
// Create a new style sheet and LoadVars object
var myVars:LoadVars = new LoadVars();
var styles = new TextField.StyleSheet();
// Location of CSS and text files to load
var txt_url = "myText.txt";
var css_url = "html_styles.css";
// Load text to display and define onLoad handler
myVars.load(txt_url);
myVars.onData = function(content) {
  storyText = content;
};
// Load CSS file and define onLoad handler:
```

```
styles.load(css_url);
styles.onLoad = function(ok) {
  if (ok) {
    // If the style sheet loaded without error,
    // then assign it to the text object,
    // and assign the HTML text to the text field.
    news_txt.styleSheet = styles;
    news_txt.text = storyText;
  }
};
```

Note: In this ActionScript, you are loading the text from an external file. For information on loading external data, see Chapter 12, "Working with External Media," on page 295.

11. Save the file as news_html.fla in the same directory that contains the CSS file you created in step 3.
12. Select Control > Test Movie to see the styles applied to the HTML text automatically.

Using styles to define new tags

If you define a new style in a style sheet, that style can be used as a tag, in the same way as you would use a built-in HTML tag. For example, if a style sheet defines a CSS style named `sectionHeading`, you can use `<sectionHeading>` as an element in any text field associated with the style sheet. This feature lets you assign arbitrary XML-formatted text directly to a text field, so that the text is automatically formatted using the rules in the style sheet.

For example, the following style sheet creates the new styles `sectionHeading`, `mainBody`, and `emphasized`:

```
.sectionHeading {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size: 18px;
  display: block
}
.mainBody {
  color: #000099;
  text-decoration: underline;
  font-size: 12px;
  display: block
}
.emphasized {
  font-weight: bold;
  display: inline
}
```

You could then populate a text field associated with that style sheet with the following XML-formatted text:

```
<sectionHeading>This is a section</sectionHeading>
<mainBody>This is some main body text,
with one <emphasized>emphatic</emphasized> word.
</mainBody>
```

An example of using styles with XML

In this section, you can create a FLA file that has XML-formatted text. You'll create a style sheet using ActionScript, rather than importing styles from a CSS file as shown in "An example of using styles with HTML" on page 232

To format XML with a style sheet:

1. In Flash, create a FLA file.
2. Using the Text tool, create a text field approximately 400 pixels wide and 300 pixels high.
3. Open the Property inspector (Window > Properties), and select the text field.
4. In the Property inspector, select Dynamic Text from the Text Type menu, select Multiline from the Line Type menu, select the Render Text as HTML option, and type `news_txt` in the Instance Name text box.
5. On Layer 1 in the Timeline (Window > Timeline), select the first frame.
6. To create the style sheet object, open the Actions panel (Window > Development Panels > Actions), and add the following code to the Actions panel:

```
var xmlStyles = new TextField.StyleSheet();
xmlStyles.setStyle("mainBody", {
  color:'#000000',
  fontFamily:'Arial,Helvetica,sans-serif',
  fontSize:'12',
  display:'block'
});
xmlStyles.setStyle("title", {
  color:'#000000',
  fontFamily:'Arial,Helvetica,sans-serif',
  fontSize:'18',
  display:'block',
  fontWeight:'bold'
});
xmlStyles.setStyle("byline", {
  color:'#666600',
  fontWeight:'bold',
  fontStyle:'italic',
  display:'inline'
});
xmlStyles.setStyle("a:link", {
  color:'#FF0000'
});
xmlStyles.setStyle("a:hover", {
  textDecoration:'underline'
});
```

This code creates a new style sheet object named `xmlStyles` that defines styles by using the `setStyle()` method. The styles exactly match the ones you created in an external CSS file earlier in this chapter.

7. To create the XML text to assign to the text field, open a text editor and enter the following text into a new document:

```
<story><title>Flash Player now supports CSS</title><mainBody><byline>San
  Francisco, CA</byline>--Macromedia Inc. announced today a new version of
  Flash Player that supports Cascading Style Sheets (CSS) text styles. For
  more information, visit the <a href="http://www.macromedia.com">Macromedia
  Flash website</a></mainBody></story>
```

Note: If you copy and paste this text string, make sure that you remove any line breaks that might have been added to the text file.

8. Save the text file as myStory.xml.
9. In Flash, add the following code in the Actions panel, following the code in step 6. This code loads the myStory.xml document, assigns the style sheet object to the text field's `styleSheet` property, and assigns the XML text to the text field:

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.load("myStory.xml");
myXML.onLoad = function(success) {
  if(success){
    storyText = myXML;
    news_txt.styleSheet = xmlStyles;
    news_txt.text = storyText;
  }else{
    trace("Error loading XML.");
  }
};
```

Note: You are loading XML data from an external file in this ActionScript. For information on loading external data, see Chapter 12, "Working with External Media," on page 295.

10. Save the file as news_xml.fla in the same folder as myStory.txt.
11. Run the movie (Control > Test Movie) to see the styles automatically applied to the text in the text field.

Using HTML-formatted text

Flash Player supports a subset of standard HTML tags such as `<p>` and `<li>` that you can use to style text in any dynamic or input text field. Text fields in Flash Player 7 and later also support the `<img>` tag, which lets you embed JPEG files, SWF files, and movie clips in a text field. Flash Player automatically wraps text around images embedded in text fields in much the same way that a web browser wraps text around embedded images in an HTML page. For more information, see "Embedding images, SWF files, and movie clips in text fields" on page 242.

Flash Player also supports the `<textformat>` tag, which lets you apply paragraph formatting styles of the TextFormat class to HTML-enabled text fields. For more information, see "Using the TextFormat class" on page 224.

For more information on HTML-formatted text, see the following topics:

- "Overview of using HTML-formatted text" on page 237
- "Supported HTML tags" on page 237
- "Embedding images, SWF files, and movie clips in text fields" on page 242

Overview of using HTML-formatted text

To use HTML in a text field, you must enable the text field's HTML formatting either by selecting the Render Text as HTML option in the Property inspector or by setting the text field's `html` property to `true`. To insert HTML into a text field, use the `TextField.htmlText` property.

For example, the following code enables HTML formatting for a text field named `headline_txt` and then assigns some HTML to the text field.

```
headline_txt.html = true;
headline_txt.htmlText = "<font face='Times New Roman' size='24'>This is how
  you assign HTML text to a text field.</font>";
```

Attributes of HTML tags must be enclosed in double (") or single (') quotation marks. Attribute values without quotation marks can produce unexpected results, such as improper rendering of text. For example, the following HTML snippet cannot be rendered properly by Flash Player because the value assigned to the `align` attribute (`left`) is not enclosed in quotation marks:

```
textField.htmlText = "<p align=left>This is left-aligned text</p>";
```

If you enclose attribute values in double quotation marks, you must *escape* the quotation marks (`\"`). Either of the following examples are acceptable:

```
textField.htmlText = "<p align='left'>This uses single quotes</p>";
textField.htmlText = "<p align=\"left\">This uses escaped double quotes</p>";
```

It's not necessary to escape double quotation marks if you're loading text from an external file; it's necessary only if you're assigning a string of text in ActionScript.

Supported HTML tags

This section lists the built-in HTML tags supported by Flash Player. You can also create new styles and tags using CSS; see "Formatting text with Cascading Style Sheets" on page 226.

Anchor tag (<a>)

The `<a>` tag creates a hypertext link and supports the following attributes:

- `href` Specifies the URL of the page to load in the browser. The URL can be either absolute or relative to the location of the SWF file that is loading the page. An example of an absolute reference to a URL is `http://www.macromedia.com`; an example of a relative reference is `/index.html`.
- `target` Specifies the name of the target window where you load the page. Options include `_self`, `_blank`, `_parent` and `_top`. The `_self` option specifies the current frame in the current window, `_blank` specifies a new window, `_parent` specifies the parent of the current frame, and `_top` specifies the top-level frame in the current window.

For example, the following HTML code creates the link “Go home,” which opens www.macromedia.com in a new browser window:

```
urlText_txt.htmlText = “<a href=’http://www.macromedia.com’ target=’_blank’>Go
  home</a>”;
```

You can use the special `asfunction` protocol to cause the link to execute an ActionScript function in a SWF file instead of opening a URL. For more information on the `asfunction` protocol, see `asfunction` in *Flash ActionScript Language Reference.*

You can also define `a:link`, `a:hover`, and `a:active` styles for anchor tags by using style sheets. See “Styling built-in HTML tags” on page 231.

Bold tag (<b>)

The `<b>` tag renders text as bold, as shown in the following example:

```
text3_txt.htmlText = "He was <b>ready</b> to leave!";
```

A bold typeface must be available for the font used to display the text.

Break tag (
)

The `<br>` tag creates a line break in the text field. In the following example, the line breaks between sentences:

```
text1_txt.htmlText = "The boy put on his coat. <br>His coat was <font
  color='#FF0033'>red</font> plaid.";
```

The closing `</br>` tag is optional, but it is good practice to include it.

Font tag (<font>)

The `<font>` tag specifies a font or list of fonts to display the text.

The font tag supports the following attributes:

- `color` Only hexadecimal color (`#FFFFFF`) values are supported. For example, the following HTML code creates red text:

  ```
  myText_txt.htmlText = "<font color='#FF0000'>This is red text</font>";
  ```

- `face` Specifies the name of the font to use. As shown in the following example, you can specify a list of comma-delimited font names, in which case Flash Player selects the first available font:

  ```
  myText_txt.htmlText = "<font face='Times, Times New Roman'>Displays as
    either Times or Times New Roman...</font>";
  ```

  If the specified font is not installed on the user’s computer system or isn’t embedded in the SWF file, Flash Player selects a substitute font.

  For more information on embedding fonts in Flash applications, see `TextField.embedFonts` in *Flash ActionScript Language Reference* and “Setting dynamic and input text options” in *Using Flash*.

- `size` Specifies the size of the font, in pixels, as shown in the following example:

```
myText_txt.htmlText = "<font size='24' color='#0000FF'>This is blue, 24-
  point text</font>";
```

  You can also use relative point sizes instead of a pixel size, such as +2 or -4.

Image tag (<img>)

The `<img>` tag lets you embed external JPEG files, SWF files, and movie clips inside text fields and TextArea component instances. Text automatically flows around images you embed in text fields or components. This tag is supported only in dynamic and input text fields that are multiline and wrap their text.

To create a multiline text field with word wrapping, do one of the following:

- In the Flash authoring environment, select a text field on the Stage and then, in the Property inspector, select Multiline from the Text Type menu.
- For a text field created at runtime with `MovieClip.createTextField()`, set the new text field instance's `TextField.multiline` and `TextField.wordWrap` properties to `true`.

The `<img>` tag has one required attribute, `src`, which specifies the path to a JPEG file, a SWF file, or the linkage identifier of a movie clip symbol in the library. All other attributes are optional.

The `<img>` tags supports the following attributes:

- `src` Specifies the URL to a JPEG or SWF file, or the linkage identifier for a movie clip symbol in the library. This attribute is required; all other attributes are optional. External files (JPEG and SWF files) do not show until they have downloaded completely.

  Note: Flash Player supports non-progressive JPEG files, but does not support progressive JPEG files.

- `id` Specifies the name for the movie clip instance (created by Flash Player) that contains the embedded JPEG file, SWF file, or movie clip. This is useful if you want to control the embedded content with ActionScript.
- `width` The width of the image, SWF file, or movie clip being inserted, in pixels.
- `height` The height of the image, SWF file, or movie clip being inserted, in pixels.
- `align` Specifies the horizontal alignment of the embedded image within the text field. Valid value are `left` and `right`. The default value is `left`.
- `hspace` Specifies the amount of horizontal space that surrounds the image where no text appears. The default value is 8.
- `vspace` Specifies the amount of vertical space that surrounds the image where no text appears. The default value is 8.

For more information and examples of using the `<img>` tag, see "Embedding images, SWF files, and movie clips in text fields" on page 242.

Italic tag (<i>)

The <i> tag displays the tagged text in italics, as shown in the following code:

```
That is very <i>interesting</i>.
```

This code example would render as follows:

That is very *interesting*.

An italic typeface must be available for the font used.

List item tag (<li>)

The <li> tag places a bullet in front of the text that it encloses, as shown in the following code:

```
Grocery list:
<li>Apples</li>
<li>Oranges</li>
<li>Lemons</li>
```

This code example would render as follows:

Grocery list:

- Apples
- Oranges
- Lemons

Note: Ordered and unordered lists (<ol> and <ul> tags) are not recognized by Flash Player, so they do not modify how your list is rendered. All list items use bullets.

Paragraph tag (<p>)

The <p> tag creates a new paragraph. It supports the following attributes:

- `align` Specifies alignment of text within the paragraph; valid values are `left`, `right`, and `center`.
- `class` Specifies a CSS style class defined by an TextField.StyleSheet object. (For more information, see "Using style classes" on page 231.)

  The following example uses the `align` attribute to align text on the right side of a text field.

  ```
  myText_txt.htmlText = "<p align='right'>This text is aligned on the right
    side of the text field</p>";
  ```

  The following example uses the `class` attribute to assign a text style class to a <p> tag:

  ```
  var myStyleSheet = new TextField.StyleSheet();
  myStyleSheet.setStyle("body", {color:'#99CCFF', fontSize:'18'});
  this.createTextField("test", 10, 0, 0, 300, 100);
  test.styleSheet = myStyleSheet;
  test.htmlText = "<p class='body'>This is some body-styled text.</p>.";
  ```

Span tag ()

The tag is available only for use with CSS text styles. (For more information, see "Formatting text with Cascading Style Sheets" on page 226.) It supports the following attribute:

- class Specifies a CSS style class defined by an TextField.StyleSheet object. For more information on creating text style classes, see "Using style classes" on page 231.

Text format tag (<textformat>)

The <textformat> tag lets you use a subset of paragraph formatting properties of the TextFormat class within HTML text fields, including line leading, indentation, margins, and tab stops. You can combine <textformat> tags with the built-in HTML tags.

The <textformat> tag has the following attributes:

- blockindent Specifies the block indentation in points; corresponds to TextFormat.blockIndent. (See TextFormat.blockIndent in *Flash ActionScript Language Reference*.)
- indent Specifies the indentation from the left margin to the first character in the paragraph; corresponds to TextFormat.indent. (See TextFormat.indent in *Flash ActionScript Language Reference*.)
- leading Specifies the amount of leading (vertical space) between lines; corresponds to TextFormat.leading. (See TextFormat.leading in *Flash ActionScript Language Reference*.)
- leftmargin Specifies the left margin of the paragraph, in points; corresponds to TextFormat.leftMargin. (See TextFormat.leftMargin in *Flash ActionScript Language Reference*.)
- rightmargin Specifies the right margin of the paragraph, in points; corresponds to TextFormat.rightMargin. (See TextFormat.rightMargin in *Flash ActionScript Language Reference*.)
- tabstops Specifies custom tab stops as an array of non-negative integers; corresponds to TextFormat.tabStops. (See TextFormat.tabStops in *Flash ActionScript Language Reference*.)

The following code example uses the tabstops attribute of the <textformat> tag to create a table of data with boldfaced row headers:

Name	Age	Department
Tim	32	Finance
Edwin	46	Marketing

To create a formatted table of data using tab stops:

1. Using the Text tool, create a dynamic text field that's approximately 300 pixels wide and 100 pixels high.
2. In the Property inspector, name the instance table_txt, select Multiline from the Line Type pop-up menu, and select the Render Text as HTML option.
3. In the Timeline, select the first frame on Layer 1.

4. Open the Actions panel (Window > Development Panels > Actions), and enter the following code in the Actions panel:

```
//Creates column headers, formatted in bold, separated by tabs
var rowHeaders = "<b>Name\tAge\tDepartment</b>";

//Creates rows with data
var row_1 = "Tim\t32\tFinance";
var row_2 = "Edwin\t46\tMarketing";

//Sets two tabstops, to 50 and 100 points
table_txt.htmlText = "<textformat tabstops='[50,100]'>";
table_txt.htmlText += rowHeaders;
table_txt.htmlText += row_1;
table_txt.htmlText += row_2 ;
table_txt.htmlText += "</textformat>";
```

The use of the tab character escape sequence (`\t`) adds tabs between each column in the table.

5. Select Control > Test Movie to view the formatted table.

Underline tag (<u>)

The `<u>` tag underlines the tagged text, as shown in the following code:

```
This is <u>underlined</u> text.
```

This code would render as follows:

This is underlined text.

Embedding images, SWF files, and movie clips in text fields

In Flash Player 7 and later, you can use the `<img>` tag to embed JPEG files, SWF files, and movie clips inside dynamic and input text fields, and TextArea component instances. (For a full list of attributes for the `<img>` tag, see "Image tag (<img>)" on page 239.)

Flash displays media embedded in a text field at full size. To specify the dimensions of the media you are embedding, use the `<img>` tags's `height` and `width` attributes. (See "Specifying height and width values" on page 243.)

In general, an image embedded in a text field appears on the line following the `<img>` tag. However, when the `<img>` tag is the first character in the text field, the image appears on the first line of the text field.

Embedding SWF and JPEG files

To embed a JPEG or SWF file in a text field, specify the absolute or relative path to the JPEG or SWF file in the `<img>` tag's `src` attribute. For example, the following code inserts a JPEG file that's located in the same directory as the SWF file:

```
textField_txt.htmlText = "<p>Here's a picture from my last vacation:<img
  src='beach.jpg'>";
```

Embedding movie clip symbols

To embed a movie clip symbol in a text field, you specify the symbol's linkage identifier for the `<img>` tag's `src` attribute. (For information on defining a linkage identifier, see "Attaching a movie clip symbol to the Stage" on page 212.)

For example, the following code inserts a movie clip symbol with the linkage identifier `symbol_ID` into a dynamic text field with the instance name `textField_txt`.

```
textField_txt.htmlText = "<p>Here's a movie clip symbol:<img
  src='symbol_ID'>";
```

In order for an embedded movie clip to display properly and completely, the registration point for its symbol should be at point (0,0).

Specifying height and width values

If you specify `width` and `height` attributes for an `<img>` tag, space is reserved in the text field for the JPEG file, SWF file, or movie clip. After a JPEG or SWF file has downloaded completely, it appears in the reserved space. Flash scales the media up or down, according to the `height` and `width` values. You must enter both `height` and `width` attributes to scale the image.

If you don't specify `height` and `width` values, no space is reserved for the embedded media. After a JPEG or SWF file has downloaded completely, Flash inserts it into the text field at full size and rebreaks text around it.

Note: If you are dynamically loading your images into a text field containing text, it is good practice to specify the width and height of the original image so the text properly wraps around the space you reserve for your image.

Controlling embedded media with ActionScript

Flash Player creates a new movie clip for each `<img>` tag and embeds that movie clip within the TextField object. The `<img>` tag's `id` attribute lets you assign an instance name to the movie clip that is created. This lets you control that movie clip with ActionScript.

The movie clip created by Flash Player is added as a child movie clip to the text field that contains the image.

For example, the following code embeds a SWF file named animation.swf in the text field named `textField_txt` on level 0 and assigns the instance name `animation_mc` to the movie clip that contains the SWF file:

```
_level0.textField_txt.htmlText = "Here's an interesting animation: <img
  src='animation.swf' id='animation_mc'>";
```

In this case, the fully qualified path to the newly created movie clip is `_level0.textField_txt.animation_mc`. For example, you could place the following code after the previous code (on the same Timeline as `textField_txt`) for a button called stop_btn that stops the playhead of the embedded SWF file:

```
stop_btn.onRelease = function() {
  _level0.textField_txt.animation_mc.stop();
};
```

Making hypertext links out of embedded media

To make a hypertext link out of an embedded JPEG file, SWF file, or movie clip, enclose the `<img>` tag in an `<a>` tag:

```
textField.htmlText = "Click the image to return home<a href='home.htm'><img
  src='home.jpg'></a>";
```

When the mouse pointer is over an image, SWF file, or movie clip that is enclosed by `<a>` tags, the pointer turns into a "pointing hand" icon, the same as it does with standard hypertext links. Interactivity, such as mouse clicks and keypresses, does not register in SWF files and movie clips that are enclosed by `<a>` tags.

Creating scrolling text

There are several ways to create scrolling text in Flash. You can make dynamic and input text fields scrollable by selecting the Scrollable option in the Text menu or the context menu, or by Shift-double-clicking the text field handle.

You can use the `scroll` and `maxscroll` properties of the TextField object to control vertical scrolling and the `hscroll` and `maxhscroll` properties to control horizontal scrolling in a text block. The `scroll` and `hscroll` properties specify the current vertical and horizontal scrolling positions, respectively; you can read and write these properties. The `maxscroll` and `maxhscroll` properties specify the maximum vertical and horizontal scrolling positions, respectively; you can only read these properties.

The TextArea component provides an easy way to create scrolling text fields with a minimum amount of scripting. For more information, see "TextArea class" in *Using Components*.

To create a scrollable dynamic text block:

Do one of the following:

- Shift-double-click the handle on the dynamic text field.
- Select the dynamic text field with the Selection tool, and select Text > Scrollable.
- Select the dynamic text field with the Selection tool. Right-click (Windows) or Control-click (Macintosh) the dynamic text field, and select Text > Scrollable.

To use the scroll property to create scrolling text:

1. Do one of the following:
   - Use the Text tool to drag a text field on the Stage. Assign the text field the instance name `textField_txt` in the Property inspector.
   - Use ActionScript to create a text field dynamically with the `MovieClip.createTextField()` method. Assign the text field the instance name `textField_txt` as a parameter of the method.

   Note: If you are *not* dynamically loading text into the SWF file, select Text › Scrollable from the main menu.

2. Create an Up button and a Down button, or select Window > Other Panels > Common Libraries > Buttons, and drag buttons to the Stage.

   You will use these buttons to scroll the text up and down.

3. Select the Down button on the Stage and type `down_btn` into the Instance Name text box.

4. Select the Up button on the Stage, and type `up_btn` into the Instance Name text box.

5. Select Frame 1 on the Timeline, and, in the Actions panel (Window > Development Panels > Actions), enter the following code to scroll the text down in the text field:

```
down_btn.onPress = function() {
  textField_txt.scroll += 1;
};
```

6. Following the ActionScript in step 5, enter the following code to scroll the text up:

```
up_btn.onPress = function() {
  textField_txt.scroll -= 1;
};
```

   Any text that loads into the `textField_txt` text field can be scrolled using the up and down buttons.

CHAPTER 10
Creating Custom Classes with ActionScript 2.0

ActionScript 2.0 is a restructuring of the ActionScript language that provides several powerful new programming features found in other programming languages, such as Java. ActionScript 2.0 encourages program structures that are reusable, scalable, robust, and maintainable. It also decreases development time by providing users thorough coding assistance and debugging information. ActionScript 2.0 conforms more closely to the ECMA-262 Edition 3 standard (see www.ecma-international.org/publications/standards/Ecma-262.htm). ActionScript 2.0 is available in Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004.

The main features of ActionScript 2.0 include the following:

Familiar object-oriented programming (OOP) model The primary feature of ActionScript 2.0 is a familiar model for creating object-oriented programs. ActionScript 2.0 implements several object-oriented concepts and keywords such as *class*, *interface*, and *packages* that will be familiar to you if you've programmed with Java.

The OOP model provided by ActionScript 2.0 is a "syntactic formalization" of the prototype chaining method used in previous versions of Macromedia Flash to create objects and establish inheritance. With ActionScript 2.0, you can create custom classes and extend Flash's built-in classes.

Strict data typing ActionScript 2.0 also lets you explicitly specify data types for variables, function parameters, and function return types. For example, the following code declares a variable named `userName` of type String (a built-in ActionScript data type, or class).

```
var userName:String = "";
```

Compiler warnings and errors The previous two features (OOP model and strict data typing) enable the authoring tool and compiler to provide compiler warnings and error messages that help you find bugs in your applications faster than was previously possible in Flash.

When you use ActionScript 2.0, make sure that the publish settings for the FLA file specify ActionScript 2.0. This is the default for files created in Flash MX 2004. However, if you open an older FLA file that uses ActionScript 1 and begin rewriting it in ActionScript 2.0, change the publish settings of the FLA file to ActionScript 2.0. If you don't, your FLA file will not compile correctly, and no errors will be generated.

This section contains code examples that you can use to become familiar with creating classes in ActionScript 2.0. If you're not familiar with ActionScript 2.0 scripting, see Chapter 2, "ActionScript Basics," on page 23 and Chapter 3, "Using Best Practices," on page 65.

Principles of object-oriented programming

This section provides a brief introduction to principles involved in developing object-oriented programs. These principles are described in more depth in the rest of this chapter, along with details on how they are implemented in Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004.

To learn about object-oriented programming principles, see the following topics:

Objects

Think of a real-world object, such as a cat. A cat could be said to have properties (or states), such as name, age, and color; a cat also has behaviors such as sleeping, eating, and purring. In the world of object-oriented programming, objects also have properties and behaviors. Using object-oriented techniques, you can model a real-world object (such as a cat) or a more abstract object (such as a chemical process).

Note: The word "behaviors" is used generically here and does not refer to the Behaviors development panel in the Flash interface.

Classes and class members

Continuing with the real-world analogy, consider that there are cats of different colors, ages, and names, with different ways of eating and purring. But despite their individual differences, all cats are members of the same category, or in object-oriented programming terms, the same class: the class of *cats*. In object-oriented programming terminology, each individual cat is said to be an instance of the cat class.

Likewise, in object-oriented programming, a *class* defines a blueprint for a type of object. The characteristics and behaviors that belong to a class are jointly referred to as *members* of that class. The characteristics (in the cat example; name, age, and color) are called *properties* of the class and are represented as variables; the behaviors (eating, sleeping) are called *methods* of the class and are represented as functions.

In ActionScript, you define a class with the `class` statement (see "Creating and using classes" on page 254).

Inheritance

One of the primary benefits of object-oriented programming is that you can create *subclasses* of a class; the subclass then *inherits* all the properties and methods of the *superclass.* The subclass typically defines additional methods and properties, or *extends* the superclass. Subclasses can also *override* (provide their own definitions for) methods inherited from a superclass.

For example, you might create a Mammal class that defines certain properties and behaviors common to all mammals. You could then create a Cat subclass that extends the Mammal class. Using subclasses lets you reuse code, so that instead of re-creating all the code common to both classes you can simply extend an existing class. Another subclass, say, the Siamese class, could extend the Cat class, and so on. In a complex application, determining how to structure the hierarchy of your classes is a large part of the design process.

In ActionScript, you use the `extends` keyword to establish inheritance between a class and its superclass. For more information, see "Creating subclasses" on page 258.

Interfaces

Interfaces in object-oriented programming can be described as classes whose methods are not implemented (defined). Another class can implement the methods declared by the interface.

An interface can also be thought of as a "programming contract" that can be used to enforce relationships between otherwise unrelated classes. For example, suppose you are working with a team of programmers, each of whom is working on a different part (class) of the same application. While designing the application, you agree on a set of methods that the different classes will use to communicate. So you create an interface that declares these methods, their parameters, and their return types. Any class that implements this interface must provide definitions for those methods; otherwise, a compiler error will result.

You can also use interfaces to provide a limited form of *multiple inheritance,* which is not allowed in ActionScript 2.0. In multiple inheritance, a class extends more than one class. For example, in C++, the Cat class could extend the Mammal class as well as a Playful class, which has methods chaseTail and eatCatNip. As with Java, ActionScript 2.0 does not allow a class to extend multiple classes directly but does allow a class to extend a single class and implement multiple interfaces. So, you could create a Playful interface that declares the chaseTail and eatCatNip methods. A Cat class, or any other class, could then implement this interface and provide definitions for those methods.

Unlike Java interfaces, ActionScript interfaces exist at runtime, which allows type casting. For more information, see "Creating an interface" on page 261.

Encapsulation

In elegant object-oriented design, objects are seen as "black boxes" that contain, or *encapsulate*, functionality. A programmer should be able to interact with an object by knowing only its properties, methods, and events (its programming interface), without knowing the details of its implementation. This approach enables programmers to think at higher levels of abstraction and provides an organizing framework for building complex systems.

Encapsulation is why ActionScript 2.0 includes, for example, member access control, so details of the implementation can be made private and invisible to code outside an object. The code outside the object is forced to interact with the object's programming interface rather than with the implementation details. This approach provides some important benefits; for example, it lets the creator of the object change the object's implementation without requiring any changes to code outside of the object, as long as the programming interface doesn't change.

Polymorphism

Object-oriented programming lets you express differences between individual classes using a technique called *polymorphism,* by which classes can override methods of their superclasses and define specialized implementations of those methods.

For example, you might start with a class called Mammal that has `play()` and `sleep()` methods. You then create Cat, Monkey, and Dog subclasses to extend the Mammal class. The subclasses override the `play()` method from the Mammal class, to reflect the habits of those particular kinds of animals. Monkey implements the `play()` method to swing from trees; Cat implements the `play()` method to pounce at a ball of yarn; Dog implements the `play()` method to fetch a ball. Because the `sleep()` functionality is similar between the animals, you would use the superclass implementation.

Using classes: a simple example

For those who are new to object-oriented programming, this section describes the workflow involved in creating and using classes in Flash and walks you through a simple hands-on example. You can also look at another simple example that describes how to assign a custom class to a movie clip symbol; see "Assigning a class to a movie clip symbol" on page 218.

At a minimum, the workflow for creating classes involves the following steps:

1. Defining a class in an external ActionScript class file.
2. Saving the class file to a designated classpath directory (a location where Flash looks for classes) or at least in the same directory as the application's FLA file. Creating an instance of the class in another script, either in a Flash (FLA) document or an external script file, or creating a subclass based on the original class.

This section also discusses a feature in ActionScript 2.0 called *strict data typing*, which lets you specify the data type for a variable, function parameter, or function return type.

Although this section discusses only classes, the general workflow is the same for using interfaces. For more information, see "Creating and using interfaces" on page 261.

The following sections contains code examples that you can use to become familiar with creating classes in ActionScript 2.0. If you're not familiar with ActionScript 2.0 scripting, see "ActionScript Basics" on page 23 and "Using Best Practices" on page 65.

For more information on using classes, see the following topics:

- "Creating a class file" on page 251
- "Creating an instance of the Person class" on page 253

Creating a class file

To create a class, you must first create an external ActionScript (AS) file. Classes (and interfaces) can be defined only in external script files. You can't define a class in a script attached to a frame or button in a Flash document (FLA file). To create an external AS file, use the ActionScript editor included with Flash Professional or your preferred code or text editor.

Note: ActionScript code in external files is compiled into a SWF file when you publish, export, test, or debug a FLA file. Therefore, if you make any changes to an external file, you must save the file and recompile any FLA files that use it.

In the following steps, you'll create a class called Person that contains two properties (`age` and `name`) and a single method (`getInfo()`) that shows the values of those properties in the Output panel.

When you create a class file, decide where you want to store the file. In the following steps, you'll save the class file and the application FLA file that uses the class file in the same directory for simplicity. However, if you want to check syntax (in Flash Professional 2004), you also need to tell Flash how it can find the file. For real applications, you would add the directory in which you store your application and class files to the Flash *classpath*. For information about classpaths, see "Understanding the classpath" on page 268 and "Modifying the classpath" on page 270. For information about organizing your project directories, see "Using packages" on page 260.

To create the class file:

1. Create a directory on your hard disk, and name it PersonFiles. This directory will contain all the files for this project.
2. (Optional) In Flash, add the PersonFiles directory to the global classpath.
3. Do one of the following:
   - Create a new file in your preferred text or code editor.
   - (Flash Professional only) Select File > New to open the New Document dialog box, select ActionScript File from the list of file types, and click OK. The Script window opens with a blank file.
4. Save the file as Person.as in the PersonFiles directory.
5. In the Script window, enter the following code (in this procedure, new code you type in each step is `bold`):

   ```
   class Person {
   }
   ```

   This is called the class *declaration*. In its most basic form, a class declaration consists of the `class` keyword, followed by the class name (Person, in this case), and then left and right curly braces (`{}`). Everything between the braces is called the class *body* and is where the class's properties and methods are defined.

   The name of the class (Person) must exactly match the name of the AS file that contains it (Person.as). This is very important; if these two names don't match exactly, including capitalization, the class won't compile.

6. To create the properties for the Person class, use the `var` keyword to define two variables named `age` and `name`, as shown in the following example:

```
class Person {

  var age:Number;
  var name:String;

}
```

Tip: By convention, class properties are defined at the top of the class body, which makes the code easier to understand, but this isn't required.

The colon syntax (`var age:Number` and `var name:String`) used in the variable declarations is an example of strict data typing. When you type a variable in this way (`var variableName:variableType`), the ActionScript 2.0 compiler ensures that any values assigned to that variable match the specified type. If the correct data type is not used in the FLA file importing this class, an error is thrown by the compiler. Using strict typing is good practice and can make debugging your scripts easier. (For more information, see "Strict data typing" on page 41.)

7. Next, you'll add a special function called a *constructor function*. In object-oriented programming, the constructor function initializes each new instance of a class.

The constructor function always has the same name as the class. To create the class's constructor function, add the following code:

```
class Person {

  var age:Number;
  var name:String;

  // Constructor function
  function Person (myName:String, myAge:Number) {
    this.name = myName;
    this.age = myAge;
  }
}
```

The `Person()` constructor function takes two parameters, `myName` and `myAge`, and assigns those parameters to the `name` and `age` properties. The two function parameters are strictly typed as String and Number, respectively. Unlike other functions, the constructor function should not declare its return type. For more information about constructor functions, see "Constructor functions" on page 255.

If you don't create a constructor function, an empty one is created automatically during compilation.

8. Last, you'll create the `getInfo()` method, which returns a preformatted string containing the values of the `age` and `name` properties. Add the `getInfo()` function definition to the class body, after the constructor function, as shown in the following code:

```
class Person {

  var age:Number;
  var name:String;
```

```
    // Constructor function
    function Person (myName:String, myAge:Number) {
      this.name = myName;
      this.age = myAge;
    }

    // Method to return property values
    function getInfo():String {
      return("Hello, my name is " + this.name + " and I'm " + this.age + "
    years old.");
    }
}
```

This code is the completed code for this class. The return value of the `getInfo()` function is strictly typed (optional, but recommended) as a string.

9. Save the file.

   If you're using Flash MX 2004 (not Flash Professional), proceed to "Creating an instance of the Person class" on page 253.

10. (Optional, Flash Professional only) Check the syntax of the class file by selecting Tools > Check Syntax, or pressing Control+T (Windows) or Command+T (Macintosh).

    If any errors are reported in the Output panel, compare the code in your script to the final code. If you can't fix the code errors, copy the completed code.

You've successfully created a class file. To continued with this example, see "Creating an instance of the Person class" on page 253.

Creating an instance of the Person class

The next step is to create an instance of the Person class in another script, such as a frame script in a Flash (FLA) document or another AS script, and assign it to a variable. To create an instance of a custom class, you use the `new` operator, the same as you would when creating an instance of a built-in ActionScript class (such as the Date or Error class). You refer to the class using its fully qualified class name or import the class; see "Importing classes" on page 271.

Continuing the example you started in "Creating a class file" on page 251, if you create a FLA file in the same directory as the class file you created, you can refer to the class file using the fully qualified class name, which is `Person`. For example, the following code creates an instance of the Person class and assigns it to the variable `newPerson`:

```
var newPerson:Person = new Person("Nate", 32);
```

This code invokes the Person class's constructor function, passing as parameters the values `"Nate"` and 32.

The `newPerson` variable is typed as a Person object. Typing your objects in this way enables the compiler to ensure that you don't try to access properties or methods that aren't defined in the class. See "Strict data typing" on page 41. (The exception is if you declare the class to be dynamic using the `dynamic` keyword. See "Creating dynamic classes" on page 259.)

To create an instance of the Person class in a Flash document:

1. In Flash, select File > New, select Flash Document from the list of document types, and click OK.
2. Save the file as createPerson.fla in the PersonFiles directory you created.

   By saving your FLA and AS files in the same directory, you can refer to the class using the fully qualified class name in step 6.
3. Select Layer 1 in the Timeline, and open the Actions panel (Window > Development Panels > Actions).
4. In the Actions panel, enter the following code:

   ```
   // assumes FLA and AS class file are in same directory
   var person_1:Person = new Person("Nate", 32);
   var person_2:Person = new Person("Jane", 28);
   trace(person_1.getInfo());
   trace(person_2.getInfo());
   ```

   This code creates two instances of the Person class, `person_1` and `person_2`, and then calls the `getInfo()` method on each instance.
5. Save your work, and then select Control > Test Movie. You should see the following text in the Output panel:

   ```
   Hello, my name is Nate and I'm 32 years old.
   Hello, my name is Jane and I'm 28 years old.
   ```

You should now have a basic understanding of how to create and use classes in your Flash documents. The rest of this chapter discusses classes and interfaces in more detail.

Creating and using classes

As discussed in "Using classes: a simple example" on page 250, a class consists of two parts: the *declaration* and the *body*. The class declaration consists minimally of the `class` statement, followed by an identifier for the class name, then left and right curly braces ({}). Everything inside the braces is the class body, as shown in the following example:

```
class className {
  // class body
}
```

You can define classes only in AS files. For example, you can't define a class in a frame script in a FLA file.

Class names must be identifiers—that is, the first character must be a letter, underscore (_), or dollar sign ($), and each subsequent character must be a letter, number, underscore, or dollar sign. The class name must exactly match the name of the AS file that contains it, including capitalization. In the following example, if you create a class called Shape, the AS file that contains the class definition must be named Shape.as:

```
// In file Shape.as
class Shape {
  // Shape class body
}
```

All AS class files that you create must be saved in one of the designated classpath directories—directories where Flash looks for class definitions when compiling scripts—that is, in the same directory where the FLA file that refers to the class is stored. (See "Understanding the classpath" on page 268.)

If you are creating multiple custom classes, use *packages* to organize your class files. A package is a directory that contains one or more class files and resides in a designated classpath directory. Class names must be fully qualified within the file in which it is declared—that is, it must reflect the directory (package) in which it is stored.

For example, a class named myClasses.education.curriculum.RequiredClass is stored in the myClasses/education/curriculum package. The class declaration in the RequiredClass.as file looks like this:

```
class myClasses.education.curriculum.RequiredClass {
}
```

For this reason, it's good practice to plan your package structure before you begin creating classes. Otherwise, if you decide to move class files after you create them, you will have to modify the class declaration statements to reflect their new location. For more information on organizing classes, see "Using packages" on page 260.

For more information on creating and using classes, see the following topics:

Constructor functions

A class's *constructor* is a special function that is called automatically when you create an instance of a class using the `new` operator. The constructor function has the same name as the class that contains it. For example, the Person class you created contained the following constructor function:

```
// Person class constructor function
function Person (myName:String, myAge:Number) {
  this.name = myName;
  this.age = myAge;
}
```

If no constructor function is explicitly declared—that is, if you don't create a function whose name matches that of the class—the compiler automatically creates an empty constructor function for you.

A class can contain only one constructor function; overloaded constructor functions are not allowed in ActionScript 2.0.

A constructor function should have no return type.

The `this` keyword is not required in ActionScript 2.0 class definitions because the compiler resolves the reference and adds it into the bytecode. However, using `this` can improve your code's readability. See "Using the this keyword" on page 96.

Creating properties and methods

A class's members consist of properties (variable declarations) and methods (function definitions). You must declare and define all properties and methods inside the class body (the curly braces [{}]); otherwise, an error will occur during compilation.

Any variable declared within a class, but outside a function, is a property of the class. In the following example, the Person class discussed in "Using classes: a simple example" on page 250 has two properties, `age` and `name`, of type Number and String, respectively:

```
class Person {
  var age:Number;
  var name:String;
}
```

Similarly, any function declared within a class is considered a method of the class. In the Person class example, you created a single method called `getInfo()`:

```
class Person {
  var age:Number;
  var name:String;
  function getInfo():String {
    // getInfo() method definition
  }
}
```

The `this` keyword is not required in ActionScript 2.0 class definitions because the compiler resolves the reference and adds it into the bytecode. However, using `this` can improve your code's readability. See "Using the this keyword" on page 96.

Controlling member access

By default, any property or method of a class can be accessed by any other class: all members of a class are *public* by default. However, in some cases you might want to protect data or methods of a class from access by other classes. You need to make those members *private* (available only to the class that declares or defines them).

You specify public or private members using the `public` or `private` member attribute. For example, the following code declares a private variable (a property) and a private method (a function). The following class (LoginClass) defines a private property named `userName` and a private method named `getUserName()`.

```
class LoginClass {
  private var userName:String;
  private function getUserName():String {
    return this.userName;
  }
  // Constructor:
  function LoginClass(user:String) {
    this.userName = user;
```

```
  }
}
```

Private members (properties and methods) are accessible only to the class that defines those members and to subclasses of that original class. Instances of the original class, or instances of subclasses of that class, cannot access privately declared properties and methods; that is, private members are accessible only within class definitions; not at the instance level.

For example, you could create a subclass of LoginClass called NewLoginClass. This subclass can access the private property (`userName`) and method (`getUserName()`) defined by LoginClass.

```
class NewLoginClass extends LoginClass {
  // can access userName and getUserName()
}
```

However, an instance of LoginClass or NewLoginClass cannot access those private members. For example, the following code, added to a frame script in a FLA file, would result in a compiler error indicating that `getUserName()` is private and can't be accessed:

```
var loginObject:LoginClass = new LoginClass("Maxwell");
var user:String = loginObject.getUserName();
```

Member access control is a compile-time-only feature; at runtime, Flash Player does not distinguish between private or public members.

The `this` keyword is not required in ActionScript 2.0 class definitions because the compiler resolves the reference and adds it into the bytecode. However, using `this` can improve your code's readability. See "Using the this keyword" on page 96.

Initializing properties inline

You can initialize properties *inline*—that is, when you declare them—with default values, as shown in the following example:

```
class Person {
  var age:Number = 50;
  var name:String = "John Doe";
}
```

When you initialize properties inline, the expression on the right side of an assignment must be a *compile-time constant*. That is, the expression cannot refer to anything that is set or defined at runtime. Compile-time constants include string literals, numbers, Boolean values, `null`, and `undefined`, as well as constructor functions for the following built-in classes: Array, Boolean, Number, Object, and String.

For example, the following class definition initializes several properties inline:

```
class CompileTimeTest {
  var foo:String = "my foo"; // OK
  var bar:Number = 5; // OK
  var bool:Boolean = true; // OK
  var name:String = new String("Jane"); // OK
  var who:String = foo; // OK, because 'foo' is a constant
  var whee:String = myFunc(); // error! not compile-time constant expression
  var lala:Number = whee; // error! not compile-time constant expression
  var star:Number = bar + 25; // OK, both 'bar' and '25' are constants
```

```
  function myFunc():String {
    return "Hello world";
  }
}
```

This rule applies only to instance variables (variables that are copied into each instance of a class), not class variables (variables that belong to the class). For more information about these kinds of variables, see "Instance and class members" on page 263.

When you initialize arrays inline, only one array is created for all instances of the class:

```
class Bar {
var foo:Array = new Array();
}
```

Creating subclasses

In object-oriented programming, a subclass can inherit the properties and methods of another class, called the superclass. To create this kind of relationship between two classes, you use the `class` statement's `extends` clause. To specify a superclass, use the following syntax:

```
class SubClass extends SuperClass {}
```

The class you specify in *`SubClass`* inherits all the properties and methods defined by the superclass. For example, you might create a Mammal class that defines properties and methods common to all mammals. To create a variation of the Mammal class, such as a Marsupial class, you would extend the Mammal class—that is, create a subclass of the Mammal class.

```
class Marsupial extends Mammal {}
```

The subclass inherits all the properties and methods of the superclass, including any properties or methods that you have declared to be private using the `private` keyword. (For more information on private variables, see "Controlling member access" on page 256.)

You can extend your own custom classes as well as many of the built-in ActionScript classes. (You cannot extend the TextField class or static classes, such as the Math, Key, and Mouse classes.) When you extend a built-in ActionScript class, your custom class inherits all the methods and properties of the built-in class.

For example, the following code defines the class JukeBox, which extends the built-in Sound class. It defines an array called `songList` and a method called `playSong()` that plays a song and invokes the `loadSound()` method, which it inherits from the Sound class.

```
class JukeBox extends Sound {
  var songList:Array = new Array("beethoven.mp3", "bach.mp3", "mozart.mp3");
  function playSong(songID:Number):Void {
    this.loadSound(songList[songID]);
  }
}
```

If you don't place a call to `super()` in the constructor function of a subclass, the compiler automatically generates a call to the constructor of its immediate superclass with no parameters as the first statement of the function. If the superclass doesn't have a constructor, the compiler creates an empty constructor function and then generates a call to it from the subclass. However, if the superclass takes parameters in its definition, you must create a constructor in the subclass and call the superclass with the required parameters.

Multiple inheritance, or inheriting from more than one class, is not allowed in ActionScript 2.0. However, classes can effectively inherit from multiple classes if you use individual `extends` statements, as shown in the following example:

```
// not allowed
class C extends A, B {}
// allowed
class B extends A {}
class C extends B {}
```

You can also use interfaces to provide a limited form of *multiple inheritance*. See "Interfaces" on page 249 and "Creating and using interfaces" on page 261.

Creating dynamic classes

By default, the properties and methods of a class are fixed. That is, an instance of a class can't create or access properties or methods that weren't originally declared or defined by the class. For example, consider a Person class that defines two properties, `name` and `age`:

```
class Person {
  var name:String;
  var age:Number;
}
```

If, in another script, you create an instance of the Person class and try to access a property of the class that doesn't exist, the compiler generates an error. For example, the following code creates a new instance of the Person class (`a_person`) and then tries to assign a value to a property named `hairColor`, which doesn't exist:

```
var a_person:Person = new Person();
a_person.hairColor = "blue"; // compiler error
```

This code causes a compiler error because the Person class doesn't declare a property named `hairColor`. In most cases, this is exactly what you want to happen. Compiler errors may not seem desirable, but they are very beneficial to programmers; good error messages help you to write correct code, by pointing out mistakes early in the coding process.

In some cases, however, you might want to add and access properties or methods of a class at runtime that aren't defined in the original class definition. The `dynamic` class modifier lets you do just that. For example, the following code adds the `dynamic` modifier to the Person class discussed previously:

```
dynamic class Person2 {
  var name:String;
  var age:Number;
}
```

Now, instances of the Person class can add and access properties and methods that aren't defined in the original class, as shown in the following example:

```
var a_person:Person2 = new Person2();
a_person.hairColor = "blue";//no compiler error because class is dynamic
trace(a_person.hairColor);
```

Subclasses of dynamic classes are also dynamic, with one exception. Subclasses of the built-in MovieClip class are not dynamic by default, even though the MovieClip class itself is dynamic. This implementation provides you with more control over subclasses of the MovieClip class, because you can choose to make your subclasses dynamic or not:

```
class A extends MovieClip {}        // A is not dynamic
dynamic class B extends A {}        // B is dynamic
class C extends B {}                // C is dynamic
class D extends A {}                // D is not dynamic
dynamic class E extends MovieClip{} // E is dynamic
```

The following built-in classes are dynamic: Array, ContextMenu, ContextMenuItem, Function, LoadVars, LocalConnection, MovieClip, SharedObject and TextField.

Using packages

When you are creating classes, organize your ActionScript class files in *packages*. A package is a directory that contains one or more class files and that resides in a designated classpath directory (see "Understanding the classpath" on page 268). A package can, in turn, contain other packages, called *subpackages*, each with its own class files.

Package names must be identifiers; that is, the first character must be a letter, underscore (_), or dollar sign ($), and each subsequent character must be a letter, number, underscore, or dollar sign.

Packages are commonly used to organize related classes. For example, you might have three related classes, Square, Circle, and Triangle, that are defined in Square.as, Circle.as, and Triangle.as. Assume that you've saved the AS files to a directory specified in the classpath, as shown in the following example:

```
// In Square.as:
class Square {}

// In Circle.as:
class Circle {}

// In Triangle.as:
class Triangle {}
```

Because these three class files are related, you might decide to put them in a package (directory) called Shapes. In this case, the fully qualified class name would contain the package path, as well as the simple class name. Package paths are denoted with dot (.) syntax, where each dot indicates a subdirectory.

For example, if you placed each AS file that defines a shape in the Shapes directory, you would need to change the name of each class file to reflect the new location, as follows:

```
// In Shapes/Square.as:
class Shapes.Square {}
```

```
// In Shapes/Circle.as:
class Shapes.Circle {}

// In Shapes/Triangle.as:
class Shapes.Triangle {}
```

To reference a class that resides in a package directory, you can either specify its fully qualified class name or import the package by using the `import` statement (see the following section).

For more information about the naming conventions for packages, see "Packages" on page 73 in Chapter 3, "Using Best Practices."

Creating and using interfaces

An interface in object-oriented programming is like a class whose methods have been declared, but otherwise don't "do" anything. That is, an interface consists of "empty" methods.

One use of interfaces is to enforce a protocol between otherwise unrelated classes. For example, suppose you're part of a team of programmers, each of whom is working on a different part—that is, a different class—of a large application. Most of these classes are unrelated, but you still need a way for the different classes to communicate. You need to define an interface, or communication protocol, to which all the classes must adhere.

One way to do this would be to create a class that defines all these methods, and then have each class *extend*, or inherit from, this superclass. But because the application consists of classes that are unrelated, it doesn't make sense to put them all into a common class hierarchy. A better solution is to create an interface that declares the methods these classes will use to communicate, and then have each class implement (provide its own definitions for) those methods.

You can usually program successfully without using interfaces. When used appropriately, however, interfaces can make the design of your applications more elegant, scalable, and maintainable.

For more information on creating and using interfaces, see the following topics:

- "Creating an interface" on page 261
- "Interfaces as data types" on page 262

Creating an interface

The process for creating an interface is the same as for creating a class. As with classes, you can define interfaces only in external AS files. You declare an interface using the `interface` keyword, followed by the interface name, and then left and right curly braces ({}), which define the body of the interface, as shown in the following example:

```
interface interfaceName {
  // interface method declarations
}
```

An interface can contain only method (function) declarations, including parameters, parameter types, and function return types.

For example, the following code declares an interface named MyInterface that contains two methods, `method_1()` and `method_2()`. The first method, `method_1()`, has no parameters and specifies a return type of `Void` (meaning it does not return a value). The second method, `method_2()`, has a single parameter of type String, and specifies a return type of Boolean.

```
interface MyInterface {
  function method_1():Void;
  function method_2(param:String):Boolean;
}
```

Interfaces cannot contain any variable declarations or assignments. Functions declared in an interface cannot contain curly braces. For example, the following interface won't compile.

```
interface BadInterface{
  // Compiler error. Variable declarations not allowed in interfaces.
  var illegalVar;

// Compiler error. Function bodies not allowed in interfaces.
  function illegalMethod(){
  }
}
```

You can also use the `extends` keyword to create subclasses of an interface:

```
interface iA extends interface iB {}
```

The rules for naming interfaces and storing them in packages are the same as those for classes; see "Creating and using classes" on page 254 and "Using packages" on page 260.

Interfaces as data types

Like a class, an interface defines a new data type. Any class that implements an interface can be considered to be of the type defined by the interface. This is useful for determining if a given object implements a given interface. For example, consider the following interface:

```
interface Movable {
  function moveUp():Void;
  function moveDown():Void;
}
```

Now consider the class Box, which implements the Movable interface.

```
class Box implements Movable {
  var xpos:Number;
  var ypos:Number;
  function moveUp():Void {
    trace("moving up");
    // method definition
  }
  function moveDown():Void {
    trace("moving down");
    // method definition
  }
}
```

In another script, such as the following code, where you create an instance of the Box class, you could declare a variable to be of the Movable type:

```
import Box;
var newBox:Movable = new Box();
```

At runtime, in Flash Player 7 and later, you can cast an expression to an interface type. Unlike Java interfaces, ActionScript interfaces exist at runtime, which allows type casting. If the expression is an object that implements the interface or has a superclass that implements the interface, the object is returned. Otherwise, `null` is returned. This is useful if you want to make sure that a particular object implements a certain interface.

For example, the following code first checks if the object name `newBox` implements the Movable interface before calling the `moveUp()` method on the object:

```
if (Movable(newBox) != null) {
  newBox.moveUp();
}
```

For more information about casting, see "Casting objects" on page 42.

Instance and class members

In object-oriented programming, members (properties or methods) of a class can be *instance members* or *class members*. Instance members are created for each instance of the class; they are defined to the prototype of the class when they are declared outside of its constructor function. In contrast, class members are created once per class. (Class members are also known as *static members*.)

To invoke an instance method or access an instance property, you reference an instance of the class. In the following example, `picture_01`, an instance of a custom class, invokes the `showInfo()` method:

```
picture_01.showInfo();
```

Class (static) members, however, are assigned to the class, not to any instance of the class. To invoke a class method or access a class property, you reference the class name, rather than a specific instance of the class, as shown in the following example:

```
ClassName.classMember;
```

For example, the ActionScript Math class consists only of static methods and properties. To call any of its methods, you don't create an instance of the Math class. Instead, you simply call the methods on the Math class itself. The following code calls the `sqrt()` method of the Math class:

```
var square_root:Number = Math.sqrt(4);
```

Creating class members

All the members (methods and properties) discussed so far in this chapter are of a type called instance members. For each instance member, there's a unique copy of that member in every instance of the class. For example, the `age` member variable of the Person class is an instance member, because each Person has a different age.

Another type of member is a class member. There is only one copy of a class member, which is used for the entire class.

The `age` property would not be a good class member, because each person has a different age. Only properties and methods that are shared by all individuals of the class should be class members.

Suppose that you want every class to have a `species` member that indicates the proper Latin name for the species that the class represents. For every Person object, the species is Homo sapiens. It would be wasteful to store a unique copy of the string `"Homo sapiens"` for every instance of the class, so this member should be a class member.

Class members are declared with the `static` modifier. For example, you could declare the `species` class member with the following code:

```
class Person
{
  static var species:String = "Homo sapiens";
  ...
}
```

You can also declare methods of a class to be static, as shown in the following code:

```
static function functionName() {
  // function body
}
```

Class (static) methods can access only class (static) properties, not instance properties. For example, the following code will result in a compiler error because the class method `getName()` references the instance variable `name`:

```
class StaticTest {
  var name:String="Ted";

  static function getName():Void {
    var local_name:String = name;
    // Error! Instance variables cannot be accessed in static functions.
  }
}
```

To solve this problem, you could either make the method an instance method or make the variable a class variable.

A common use of class members is the *Singleton design pattern*. The Singleton design pattern makes sure that a class has only one instance and provides a way of globally accessing the instance. For more information on the Singleton design pattern, see www.macromedia.com/devnet/mx/coldfusion/articles/design_patterns.html.

Often there are situations when you need exactly one object of a particular type in a system. For example, in a chess game, there is only one chessboard, and in a country, there is only one capitol city. Even though there is only one object, it is attractive to encapsulate the functionality of this object in a class. However, you might need to manage and access the one instance of that object. Using a global variable is one way to do this, but global variables are often not desirable. A better approach is to make the class manage the single instance of the object itself using class members, such as the following example:

```
class Singleton {
  private var instance:Singleton = null;
  public function doSomething():Void {
    //...
  }
  public static function getInstance():Singleton {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

The Singleton object can then be accessed using `Singleton.getInstance();`

This also means that the Singleton object is not created until it is actually needed—that is, until some other code asks for it by calling the `getInstance` method. This is typically called *lazy creation* and can help code efficiency in many circumstances.

Using class members: a simple example

One use of class (static) members is to maintain state information about a class and its instances. For example, suppose you want to keep track of the number of instances that have been created from a particular class. An easy way to do this is to use a class property that's incremented each time a new instance is created.

In the following example, you'll create a class called Widget that defines a single, static instance counter named `widgetCount`. Each time a new instance of the class is created, the value of `widgetCount` is incremented by 1 and the current value of `widgetCount` is displayed in the Output panel.

To create an instance counter using a class variable:

1. Create a new ActionScript (AS) file.
2. Add the following code to the file:

```
class Widget {
  static var widgetCount:Number = 0; // initialize class variable
  function Widget() {
    trace("Creating widget #" + widgetCount);
    widgetCount++;
  }
}
```

The `widgetCount` variable is declared as static, so it initializes to 0 only once. Each time the Widget class's constructor function is called, it adds 1 to `widgetCount` and then shows the number of the current instance that's being created.

3. Save your file as Widget.as.
4. Create a new FLA document, and save it as createWidget.fla in the same directory as Widget.as.

   In this file, you'll create new instances of the Widget class.
5. In createWidget.fla, select Layer 1 in the Timeline, and open the Actions panel (Window > Development Panels > Actions).
6. Add the following code to the Actions panel:

```
// Before you create any instances of the class,
// widgetCount is zero (0)
trace("Widget count at start: " + Widget.widgetCount);
var widget_1:Widget = new Widget();
var widget_2:Widget = new Widget();
var widget_3:Widget = new Widget();
```

7. Save your work, and then select Control > Test Movie. You should see the following text in the Output panel:

```
Widget count at start: 0
Creating widget # 0
Creating widget # 1
Creating widget # 2
```

Class members and subclasses

Class members propagate to subclasses of the superclass that defines those members. In the previous example (see "Using class members: a simple example" on page 265), you used a class property to keep track of the number of instances of the class you created. You could create a subclass of the Widget class, as shown in the following code:

```
class SubWidget extends Widget {
  function SubWidget() {
    trace("Creating subwidget # "+Widget.widgetCount);
  }
}
```

The ActionScript 2.0 compiler can resolve static member references within class definitions. In the previous example, if you don't specify the class name for the `Widget.widgetCount` property, but instead refer only to `widgetCount`, the ActionScript 2.0 compiler ascertains that the reference is actually to `Widget.widgetCount` and correctly exports that property. Similarly, if you referred to the property as `SubWidget.widgetCount`, the compiler rewrites the reference (in the bytecode, not in your AS file) as `Widget.widgetCount` because `SubWidget` is a subclass of the `Widget` class.

However, for optimal readability of your code, it is recommended that you always use explicit references in your code, as shown in the previous example, to easily identify where the definition of a static member resides.

Implicit getter/setter methods

Object-oriented programming practice discourages direct access to properties within a class. Classes typically define *getter* methods that provide read access and *setter* methods that provide write access to a given property. For example, imagine a class that contains a property called `userName`:

```
var userName:String;
```

Instead of allowing instances of the class to directly access this property (`obj.userName = "Jody"`, for example), the class might have two methods, `getUserName` and `setUserName`, that would be implemented as shown in the following example:

```
class LoginClass {

  private var userName:String;

  function LoginClass(name:String) {
    this.userName = name;
  }
  function getUserName():String {
    return this.userName;
  }
  function setUserName(name:String):Void {
    this.userName = name;
  }
}
```

As you can see, `getUserName` returns the current value of `userName`, and `setUserName` sets the value of `userName` to the string parameter passed to the method. An instance of the class would then use the following syntax to get or set the `userName` property:

```
var obj:LoginClass = new LoginClass("RickyM");
// calling "get" method
var name = obj.getUserName();
trace(name);
// calling "set" method
obj.setUserName("EnriqueI");
trace(obj.getUserName());
```

However, if you want to use a more concise syntax, use *implicit* getter/setter methods. Implicit getter/setter methods let you access class properties in a direct manner, while maintaining good OOP practice.

To define these methods, use the `get` and `set` method attributes. You create methods that get or set the value of a property, and add the keyword `get` or `set` before the method name, as shown in the following example:

```
class LoginClass2 {

  private var userName:String;

  function LoginClass2(name:String) {
    this.userName = name;
  }
  function get user():String {
```

```
    return this.userName;
  }
  function set user(name:String):Void {
    this.userName = name;
  }
}
```

A getter method must not take any parameters. A setter method must take exactly one required parameter. A setter method can have the same name as a getter method in the same scope. Getter/setter methods cannot have the same name as other properties. For example, in the example code above that defines getter and setter methods named `user`, you could not also have a property named `user` in the same class.

Unlike ordinary methods, getter/setter methods are invoked without any parentheses or arguments. For example, the following syntax could now be used to access or modify the value of `userName` with the getter/setter methods previously defined:

```
var obj:LoginClass2 = new LoginClass2("RickyM");
// calling "get" method
trace(obj.user);
// calling "set" method
obj.user = "EnriqueI";
trace(obj.user);
```

Getter/setter method attributes cannot be used in interface method declarations.

Note: Implicit getter/setter methods are syntactic shorthand for the `Object.addProperty()` method in ActionScript 1.

Understanding the classpath

In order to use a class or interface that you've defined, Flash must locate the external AS files that contain the class or interface definition. The list of directories in which Flash searches for class and interface definitions is called the *classpath*.

When you create an ActionScript class file, you need to save the file to one of the directories specified in the classpath or a subdirectory therein. (You can modify the classpath to include the desired directory path; see "Modifying the classpath" on page 270.) Otherwise, Flash won't be able to *resolve*, or locate, the class or interface specified in the script. Subdirectories that you create within a classpath directory are called *packages* and let you organize your classes. (For more information, see "Using packages" on page 260.)

To learn about classpaths, see the following sections:

- "Global and document-level classpaths" on page 268
- "How the compiler resolves class references" on page 269
- "Modifying the classpath" on page 270

Global and document-level classpaths

Flash has two classpath settings: a global classpath and a document-level classpath. The global classpath applies to external AS files and to FLA files, and is set in the Preferences dialog box (Edit > Preferences > ActionScript > ActionScript 2.0 Settings).

By default, the global classpath contains one absolute path and one relative path. The absolute path is denoted by $(LocalData)/Classes in the Preferences dialog box. The location of the absolute path is shown here:

- (Windows 2000 or Windows XP) \Documents and Settings*user*\Local Settings\Application Data\Macromedia\Flash MX 2004*language*\Configuration\Classes
- (Windows 98) \Windows\Application Data\Macromedia\Flash MX 2004\ *language*\Configuration\Classes
- (Macintosh OS X) Hard Drive/Users/Library/Application Support/Macromedia/Flash MX 2004/*language*/Configuration/Classes

Note: Do not delete the absolute global classpath. Flash uses this classpath to access built-in classes. If you accidentally delete this classpath, see "Modifying the classpath" on page 270 for information on how to reinstate it in the classpath list.

The relative path portion of the global classpath is denoted by a single dot (.) and points to the current document directory. Be aware that relative classpaths can point to different directories, depending on the location of the document being compiled or published.

The document-level classpath applies only to FLA files and is set in the Publish Settings dialog box for a particular FLA file (File > Publish Settings > Flash > ActionScript 2.0 Settings). The document-level classpath is empty by default. When you create and save a FLA file in a directory, that directory becomes a designated classpath directory.

When you create classes, in some cases you might want to store them in a directory that you then add to the list of global classpath directories: 1) If you have a set of utility classes that all your projects use; 2) If you have a particular class on which you want to Check Syntax within the external AS file (Flash Professional only). This would prevent losing custom classes if you ever uninstall and reinstall Flash, in case the default global classpath directory is deleted and overwritten and any classes stored in that directory would be lost.

For example, you might create a directory such as the following for your custom classes:

- (Windows 2000 or Windows XP) \Documents and Settings*user*\Local Settings\Application Data\Macromedia\My_Flash MX 2004\Classes
- (Windows 98) \Windows\Application Data\Macromedia\My_Flash MX 2004\Classes
- (Macintosh OS X) Hard Drive/Users/Library/Application Support/Macromedia/My_Flash MX 2004/Classes

Then, you would add this path to the list of global classpaths (see "Modifying the classpath" on page 270).

How the compiler resolves class references

When Flash attempts to resolve class references in a FLA script, it first searches the document-level classpath specified for that FLA file. If the class is not found in that classpath, or if that classpath is empty, Flash searches the global classpath. If the class is not found in the global classpath, a compiler error occurs.

In Flash Professional, when you Check Syntax while editing an AS file, the compiler looks only in the global classpath; AS files aren't associated with FLAs in Edit mode and don't have their own classpath.

Modifying the classpath

You can modify the global classpath using the Preferences dialog box. To modify the document-level classpath setting, you use the Publish Settings dialog box for the FLA file. In both cases, you can add absolute directory paths (for example, C:/my_classes) and relative directory paths (for example, ../my_classes or "."). The order of directories in the dialog box reflects the order in which they will be searched.

To modify the global classpath:

1. Select Edit > Preferences to open the Preferences dialog box.
2. Click the ActionScript tab, and then click the ActionScript 2.0 Settings button.
3. Do any of the following:
   - To add a directory to the classpath, click the Browse to Path button, browse to the directory you want to add, and click OK.

     Alternatively, click the Add New Path (+) button to add a new line to the Classpath list. Double-click the new line, type a relative or absolute path, and click OK.
   - To edit an existing classpath directory, select the path in the Classpath list, click the Browse to Path button, browse to the directory you want to add, and click OK.

     Alternatively, double-click the path in the Classpath list, type the desired path, and click OK.
   - To delete a directory from the classpath, select the path in the Classpath list and click the Remove from Path button.

Note: Do not delete the absolute global classpath (see "Global and document-level classpaths" on page 268). Flash uses this classpath to access built-in classes. If you accidentally delete this classpath, reinstate it by adding `$(LocalData)/Classes` as a new classpath.

To modify the document-level classpath:

1. Select File > Publish Settings to open the Publish Settings dialog box.
2. Click the Flash tab.
3. Click the Settings button next to the ActionScript Version pop-up menu.
4. Do any of the following:
   - To add a directory to the classpath, click the Browse to Path button, browse to the directory you want to add, and click OK.

     Alternatively, click the Add New Path (+) button to add a new line to the Classpath list. Double-click the new line, type a relative or absolute path, and click OK.
   - To edit an existing classpath directory, select the path in the Classpath list, click the Browse to Path button, browse to the directory you want to add, and click OK.

Alternatively, double-click the path in the Classpath list, type the desired path, and click OK.

- To delete a directory from the classpath, select the path in the Classpath list and click the Remove Selected Path (-) button.

Importing classes

To reference a class in another script, you must prefix the class name with the class's package path. The combination of a class's name and its package path is the class's *fully qualified class name*. If a class resides in a top-level classpath directory—not in a subdirectory in the classpath directory—then its fully qualified class name is its class name.

To specify package paths, use dot (.) notation to separate package directory names. Package paths are hierarchical, where each dot represents a nested directory. For example, suppose you create a class named Data that resides in a com/xyzzycorporation/ package in your classpath. To create an instance of that class, you could specify the fully qualified class name, as shown in the following example:

```
var dataInstance = new com.xyzzycorporation.Data();
```

You can also use the fully qualified class name to type your variables, as shown in the following example:

```
var dataInstance:com.xyzzycorporation.Data = new Data();
```

You can use the `import` statement to import packages into a script, which lets you use a class's abbreviated name rather than its fully qualified name. You can also use the wildcard character (*) to import all the classes in a package.

For example, suppose you created a class named UserClass that's included in the package directory path com/xyzzycorporation/util/users:

```
// In the file com/xyzzycorporation/util/users/UserClass.as
class com.xyzzycorporation.util.users.UserClass { ... }
```

Suppose that in another script, you imported that class using the `import` statement, as shown in the following example:

```
import com.xyzzycorporation.util.users.UserClass;
```

Later, in the same script, you could reference that class by its abbreviated name, as shown in the following example:

```
var myUser:UserClass = new UserClass();
```

You can use the wildcard character (*) to import all the classes in a given package. For example, suppose you have a package named `com.xyzzycorporation.util` that contains two ActionScript class files, Rosencrantz.as and Guildenstern.as. In another script, you could import both classes in that package using the wildcard character, as shown in the following code:

```
import com.xyzzycorporation.util.*;
```

The following example shows that you can then reference either of the classes directly in the same script:

```
var myRos:Rosencrantz = new Rosencrantz();
var myGuil:Guildenstern = new Guildenstern();
```

The `import` statement applies only to the current script (frame or object) in which it's called. If an imported class is not used in a script, the class is not included in the resulting SWF file's bytecode, and the class isn't available to any SWF files that the FLA file containing the `import` statement might load. For more information, see `import` in *Flash ActionScript Language Reference*.

Compiling and exporting classes

By default, classes used by a SWF file are packaged and exported in the SWF file's first frame. You can also specify the frame where your classes are packaged and exported. This is useful, for example, if a SWF file uses many classes that require a long time to download. If the classes are exported in the first frame, the user has to wait until all the class code has downloaded before that frame appears. By specifying a later frame in the Timeline, you could display a short loading animation in the first few frames of the Timeline while the class code in the later frame downloads.

To specify the export frame for classes for a Flash document:

1. With a FLA file open, select File > Publish Settings.
2. In the Publish Settings dialog box, click the Flash tab.
3. Click the Settings button next to the ActionScript version pop-up menu to open the ActionScript Settings dialog box.
4. In the Export Frame for Classes text box, enter the number of the frame where you want to export your class code.

   If the frame specified does not exist in the Timeline, you will get an error message when you publish your SWF file.
5. Click OK to close the ActionScript Settings dialog box, and then click OK to close the Publish Settings dialog box.

During compilation, Flash sometimes creates files with .aso extensions in the /aso subdirectory of the default global classpath directory (see "Global and document-level classpaths" on page 268). The .aso extension stands for *ActionScript object* (ASO). These files contain the compiled form of a class file.

The compiler creates ASO files for caching purposes. You might notice that your first compilation is slower than subsequent compilations. This is because only the AS files that have changed are recompiled into ASO files. For unchanged AS files, the compiler reads the already-compiled bytecode directly out of the ASO file instead of recompiling the AS file.

The ASO file format is an intermediate format developed for internal use only. It is not a documented file format and is not intended to be redistributed.

If you experience problems in which Flash appears to be compiling older versions of a file you have edited, delete the ASO files and then recompile. If you plan to delete ASO files, delete them when Flash is not performing other operations, such as checking syntax or exporting SWFs.

There is a limit to how much code you can place in a single class: The bytecode for a class definition in an exported SWF file cannot be larger than 32,767 bytes. If the bytecode is larger than that limit, a warning message appears.

You can't predict the size of the bytecode representation of a given class, but classes up to 1,500 lines usually don't go over the limit.

If your class goes over the limit, move some of the code into another class. In general, it is good OOP practice to keep classes relatively short.

Excluding classes

To reduce the size of a SWF file, you might want to exclude classes from compilation but still be able to access and use them for type checking. For example, you might want to do this if you are developing an application that uses multiple SWF files or shared libraries that access many of the same classes, and you want to avoid duplicating classes in the files. To exclude classes from compilation, create a specially named and formatted XML file and place it in the same directory as the FLA file. Name the XML file *FLA_filename*_exclude.xml, where *FLA_filename* is the name of your FLA file minus the extension. For example, if your FLA file is sellStocks.fla, the XML filename must be sellStocks_exclude.xml.

Place the following tags in the XML file:

```
<excludeAssets>
<asset name="className1"></asset>
<asset name="className2"></asset>
...
</excludeAssets>
```

The values you specify for the `name` attributes in the `<asset>` tags are the names of classes you want to exclude from the SWF file. For example, the following XML file excludes the `mx.core.UIObject` and `mx.screens.Slide` classes from the SWF file:

```
<excludeAssets>
<asset name="mx.core.UIObject"></asset>
<asset name="mx.screens.Slide"></asset>
</excludeAssets>
```

CHAPTER 11
Working with External Data

In Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004, you can use ActionScript to load data from external sources into a SWF file. You can also send data from a SWF file for processing by an application server (such as Macromedia ColdFusion MX or Macromedia JRun) or another type of server-side script, such as PHP or Perl. Macromedia Flash Player can send and load data over HTTP or HTTPS or load from a local text file. You can also create persistent TCP/IP socket connections for applications that require low latency—for example, chat applications or stock quote services.

Data that you load into or send from a SWF file can be formatted as XML (Extensible Markup Language) or as name-value pairs.

Flash Player can also send data to and receive data from its host environment—a web browser, for example—or another instance of Flash Player on the same computer or web page.

By default, a SWF file can access only data that resides in exactly the same domain (for example, www.macromedia.com). (For more information, see "Flash Player security features" on page 288.)

Sending and loading variables to and from a remote source

A SWF file is a window for capturing and displaying information, much like an HTML page. However, SWF files can stay loaded in the browser and continuously update with new information without having to reload the entire page. Using ActionScript functions and methods, you can send information to and receive information from server-side scripts, and receive information from text files and XML files.

In addition, server-side scripts can request specific information from a database and relay it to a SWF file. Server-side scripts can be written in different languages: some of the most common are CFML, Perl, ASP (Microsoft Active Server Pages), and PHP. By storing information in a database and retrieving it, you can create dynamic and personalized content for your SWF file. For example, you could create a message board, personal profiles for users, or a shopping cart that keeps track of a user's purchases.

Several ActionScript functions and methods let you pass information into and out of a SWF file. Each function or method uses a protocol to transfer information and requires information to be formatted in a certain way.

- The functions and MovieClip methods that use the HTTP or HTTPS protocol to send information in URL-encoded format are `getURL()`, `loadVariables()`, `loadVariablesNum()`, `loadMovie()`, and `loadMovieNum()`.
- The LoadVars methods that use the HTTP or HTTPS protocol to send and load information in URL-encoded format are `load()`, `send()`, and `sendAndLoad()`.
- The methods that use HTTP or HTTPS protocol to send and load information as XML are `XML.send()`, `XML.load()`, and `XML.sendAndLoad()`.
- The methods that create and use a TCP/IP socket connection to send and load information as XML are `XMLSocket.connect()` and `XMLSocket.send()`.

For more information, see the following topics:

- "Checking for loaded data" on page 276
- "Using HTTP to connect to server-side scripts" on page 277
- "Using the LoadVars class" on page 278
- "About XML" on page 280
- "Using the XML class" on page 280
- "Using the XMLSocket class" on page 284

Checking for loaded data

Each function or method that loads data into a SWF file (except `XMLSocket.send()`) is *asynchronous*: The results of the action are returned at an indeterminate time.

Before you can use loaded data in a SWF file, you must check to see if it has been loaded. For example, you can't load variables and manipulate their values in the same script because the data to manipulate doesn't exist in the file until it is loaded. In the following script, you cannot use the variable `lastSiteVisited` until you're sure that the variable has loaded from the file `myData.txt`. In the file myData.txt, you would have text similar to the following example:

```
lastSiteVisited=www.macromedia.com
```

But if you used the following code, you could not trace the data that is loading:

```
loadVariables("myData.txt", 0);
trace(lastSiteVisited);
```

Each function or method has a specific technique you can use to check data it has loaded. If you use `loadVariables()` or `loadMovie()`, you can load information into a movie clip target and use the `onData` handler to execute a script. If you use `loadVariables()` to load the data, the `onData` handler executes when the last variable is loaded. If you use `loadMovie()` to load the data, the `onData` handler executes each time a fragment of the SWF file is streamed into Flash Player.

Each function or method has a specific technique you can use to check data it has loaded. If you use `loadVariables()`, you can load information into a movie clip target and use the `MovieClip.onData` handler to execute a script; the `onData` handler executes when the last variable is loaded.

For example, the following ActionScript loads the variables from the file `myData.txt` into the movie clip `loadTarget_mc`. An `onData()` handler assigned to the `loadTarget_mc` instance uses the variable `lastSiteVisited`, which is loaded from the file `myData.txt`. The following trace actions appear only after all the variables, including `lastSiteVisited`, are loaded:

```
this.createEmptyMovieClip("loadTarget_mc", this.getNextHighestDepth());
this.loadTarget_mc.onData = function() {
  trace("Data Loaded");
  trace(this.lastSiteVisited);
};
loadVariables("myData.txt", this.loadTarget_mc);
```

If you use the `XML.load()`, `XML.sendAndLoad()`, and `XMLSocket.connect()` methods, you should define a handler that will process the data when it arrives. This handler is a property of an XML or XMLSocket object to which you assign a function you defined. The handlers are called automatically when the information is received. For the XML object, use `XML.onLoad()` or `XML.onData()`. For the XMLSocket object, use `XMLSocket.onConnect()`.

For more information, see "Using the XML class" on page 280 and "Using the XMLSocket class" on page 284. For more information on using LoadVars to send and load data that can be processed after the data is received, see "Using the LoadVars class" on page 278.

Using HTTP to connect to server-side scripts

The `loadVariables()`, `loadVariablesNum()`, `getURL()`, `loadMovie()`, `loadMovieNum()` functions and the `MovieClip.loadVariables()`, `MovieClip.loadMovie()`, and `MovieClip.getURL()` methods can communicate with server-side scripts using HTTP or HTTPS protocols. These functions and methods send all the variables from the Timeline to which the function is attached. When used as methods of the MovieClip object, `loadVariables()`, `getURL()`, and `loadMovie()` send all the variables of the specified movie clip; each function (or method) handles its response as follows:

- The `getURL()` function returns any information to a browser window, not to Flash Player.
- The `loadVariables()` methods loads variables into a specified Timeline or level in Flash Player.
- The `loadMovie()` methods loads a SWF file into a specified level or movie clip in Flash Player.

When you use `loadVariables()`, `getURL()`, or `loadMovie()`, you can specify several parameters:

- *`URL`* is the file in which the remote variables reside.
- *`Location`* is the level or target in the SWF file that receives the variables. (The `getURL()` function does not take this parameter.)

  For more information about levels and targets, see "Multiple Timelines and levels" in *Using Flash*.
- *`Variables`* sets the HTTP method, either `GET` (appends the variables to the end of the URL) or `POST` (sends the variables in a separate HTTP header), by which the variables are sent. When this parameter is omitted, Flash Player defaults to `GET`, but no variables are sent.

For example, if you want to track the high scores for a game, you could store the scores on a server and use `loadVariables()` to load them into the SWF file each time someone played the game. The function call might look like the following example:

```
loadVariables("http://www.mySite.com/scripts/high_score.cfm", _root.scoreClip,
  GET);
```

This example loads variables from the ColdFusion script called `high_score.cfm` into the movie clip instance `scoreClip` using the `GET` HTTP method.

Any variables loaded with the `loadVariables()` function must be in the standard MIME format *application/x-www-form-urlencoded* (a standard format used by CFM and CGI scripts). The file you specify in the *URL* parameter of `loadVariables()` must write out the variable and value pairs in this format so that Flash can read them.This file can specify any number of variables; variable and value pairs must be separated with an ampersand (&), and words within a value must be separated with a plus (+). For example, the following phrase defines several variables:

```
highScore1=54000&playerName1=RGoulet&highScore2=53455&playerName2=
  WNewton&highScore3=42885&playerName3=TJones
```

Note: You might need to URL-encode certain characters, such as the plus (+) or ampersand (&) characters. For more information, see www.macromedia.com/support/flash/ts/documents/url_encoding.htm.

For more information, see "Using the LoadVars class" on page 278. Also, see `loadVariables()`, `getURL()`, `loadMovie()`, and the `LoadVars class` entry in ActionScript Language Reference Help.

Using the LoadVars class

If you are publishing to Flash Player 6 or later and want more flexibility than `loadVariables()` offers, you can use the LoadVars class instead to transfer variables between a SWF file and a server.

The LoadVars class was introduced in Flash Player 6 to provide a cleaner, more object-oriented interface for the common task of exchanging CGI data with a web server. Advantages of the LoadVars class include the following:

- You don't need to create container movie clips for holding data or clutter existing movie clips with variables specific to client/server communication.
- The class interface is similar to the XML object, which provides some consistency in ActionScript. It uses the methods `load()`, `send()`, and `sendAndLoad()` to initiate communication with a server. The main difference between the LoadVars and XML classes is that the LoadVars data is a property of the LoadVars object, rather than an XML Document Object Model (DOM) tree stored in the XML object.
- The class interface is more straightforward—with methods named `load`, `send`, `sendAndLoad`—than the older loadVariables interface.
- You can get additional information about the communication, using the `getBytesLoaded` and `getBytesTotal` methods
- You can get progress information about the download of your data (although you can't access the data until it is fully downloaded).

- The callback interface is through ActionScript methods (`onLoad`) instead of the obsolete, deprecated `onClipEvent` (data) approach required for loadVariables.
- There are error notifications.
- You can add custom HTTP request headers.

You must create a LoadVars object to call its methods. This object is a container to hold the loaded data.

The following procedure shows how to use ColdFusion and the LoadVars class to send an e-mail from a SWF file.

Note: You must have ColdFusion installed on your web server for this example.

To load data with the LoadVars object:

1. Create a CFM file in Macromedia Dreamweaver or in your favorite text editor. Add the following text to the file:

```
<cfif IsDefined("Form")>
<cfmail to="#Form.emailTo#" from="#Form.emailFrom#"
  subject="#Form.emailSubject#">#Form.emailBody#</cfmail>
&result=true
<cfelse>
&result=false
</cfif>
```

2. Save the file as email.cfm, and upload it to your website.
3. In Flash, create a new document.
4. Create four input text fields on the Stage, and give them the following instance names: `emailFrom_txt`, `emailTo_txt`, `emailSubject_txt`, and `emailBody_txt`.
5. Create a dynamic text field on the Stage with the instance name `debug_txt`.
6. Drag a PushButton component instance to the Stage, and give it the instance name `submit_btn`.
7. Select Frame 1 in the Timeline, and open the Actions panel (Window > Development Panels > Actions) if it isn't already open.
8. Enter the following code in the Actions panel:

```
this.submit_btn.onRelease = function() {
  var emailResponse:LoadVars = new LoadVars();
  var email:LoadVars = new LoadVars();
  email.emailFrom = emailFrom_txt.text;
  email.emailTo = emailTo_txt.text;
  email.emailSubject = emailSubject_txt.text;
  email.emailBody = emailBody_txt.text;
  email.sendAndLoad("http://www.yoursite.com/email.cfm", emailResponse,
  "POST");
  emailResponse.onLoad = function() {
    debug_txt.text = this.result;
  };
};
```

This ActionScript creates a new LoadVars object instance, copies the values from the text fields into the instance, and then sends the data to the server. The CFM file sends the e-mail and returns a variable (`true` or `false`) to the SWF file called `result`, which appears in the `debug_txt` text field.

Note: Remember to change the URL www.yoursite.com to your own domain.

9. Save the document as sendEmail.fla, and then publish it by selecting File > Publish.
10. Upload sendEmail.swf to the same directory that contains email.cfm (the ColdFusion file you saved and uploaded in step 2).
11. View and test the SWF file in a browser.

For more information, see the "LoadVars class" entry in *Flash ActionScript Language Reference*.

About XML

Extensible Markup Language (XML) is becoming the standard for exchanging structured data in Internet applications. You can integrate data in Flash with servers that use XML technology to build sophisticated applications, such as chat or brokerage systems.

In XML, as with HTML, you use tags to specify, or *mark up*, a body of text. In HTML, you use predefined tags to indicate how text should appear in a web browser (for example, the `<b>` tag indicates that text should be bold). In XML, you define tags that identify the type of a piece of data (for example, `<password>VerySecret</password>`). XML separates the structure of the information from the way it appears, so the same XML document can be used and reused in different environments.

Every XML tag is called a *node*, or an element. Each node has a type (1, which indicates an XML element, or 3, which indicates a text node), and elements might also have attributes. A node nested in a node is called a *child node*. This hierarchical tree structure of nodes is called the XML DOM—much like the JavaScript DOM, which is the structure of elements in a web browser.

In the following example, `<portfolio>` is the parent node; it has no attributes and contains the child node `<holding>`, which has the attributes `symbol`, `qty`, `price`, and `value`:

```
<portfolio>
  <holding symbol="rich"
    qty="75"
    price="245.50"
    value="18412.50" />
</portfolio>
```

For more information on XML, see www.w3.org/XML.

Using the XML class

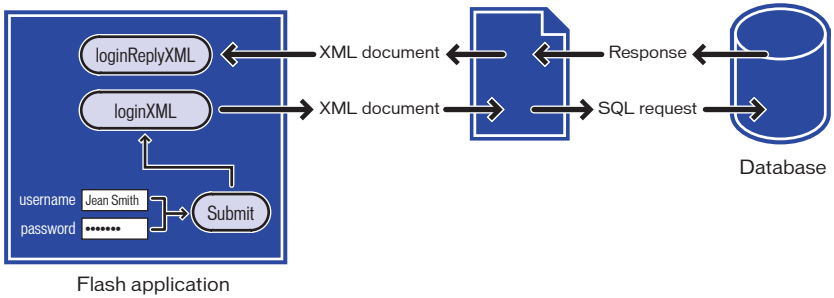
The methods of the ActionScript XML class (for example, `appendChild()`, `removeNode()`, and `insertBefore()`) let you structure XML data in Flash to send to a server and manipulate and interpret downloaded XML data.

The following XML class methods send and load XML data to a server by using the HTTP `POST` method:

- The `load()` method downloads XML from a URL and places it in an ActionScript XML object.
- The `send()`method encodes the XML object into an XML document and sends it to a specified URL using the `POST` method. If specified, a browser window displays returned data.
- The `sendAndLoad()` method sends an XML object to a URL. Any returned information is placed in an ActionScript XML object.

For example, you could create a brokerage system that stores all its information (user names, passwords, session IDs, portfolio holdings, and transaction information) in a database.

The server-side script that passes information between Flash and the database reads and writes the data in XML format. You can use ActionScript to convert information collected in the SWF file (for example, a user name and password) to an XML object and then send the data to the server-side script as an XML document. You can also use ActionScript to load the XML document that the server returns into an XML object to be used in the SWF file.



The flow and conversion of data between a SWF file, a server-side script, and a database

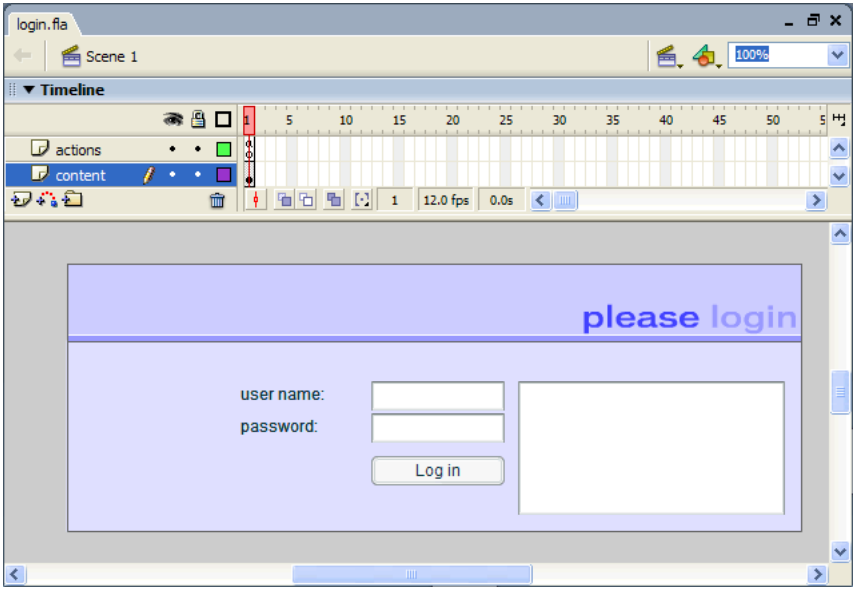
The password validation for the brokerage system requires two scripts: a function defined on Frame 1, and a script that creates and then sends the XML objects created in the document.

When a user enters information into text fields in the SWF file with the variables `username` and `password`, the variables must be converted to XML before being passed to the server. The first section of the script loads the variables into a newly created XML object called `loginXML`. When a user clicks a button to log in, the `loginXML` object is converted to a string of XML and sent to the server.

The following ActionScript is placed on the Timeline and is used to send XML-formatted data to the server. To understand this script, read the commented lines (indicated by the characters `//`):

```
//ignore XML white space
XML.prototype.ignoreWhite = true;
// Construct an XML object to hold the server's reply
var loginReplyXML:XML = new XML();
// this function triggers when an XML packet is received from the server.
loginReplyXML.onLoad = function(success:Boolean) {
  if (success) {
```

```
      //(optional) Create two text fields for status/debugging
      //status_txt.text = this.firstChild.attributes.status;
      //debug_txt.text = this.firstChild;
      switch (this.firstChild.attributes.status) {
      case 'OK' :
        _global.session = this.firstChild.attributes.session;
        trace(_global.session);
        gotoAndStop("welcome");
        break;
      case 'FAILURE' :
        gotoAndStop("loginfailure");
        break;
      default :
        // this should never happen
        trace("Unexpected value received for STATUS.");
      }
    } else {
      // trace(“an error occurred.”);
    }
};
// this function triggers when the login_btn is clicked
login_btn.onRelease = function() {
  var loginXML:XML = new XML();
  //create XML formatted data to send to the server
  var loginElement:XMLNode = loginXML.createElement("login");
  loginElement.attributes.username = username_txt.text;
  loginElement.attributes.password = password_txt.text;
  loginXML.appendChild(loginElement);
  //send the XML formatted data to the server
  loginXML.sendAndLoad("http://www.flash-mx.com/mm/main.cfm", loginReplyXML);
};
```



You can test this code by using a user name of `JeanSmith` and the password `VerySecret`. The first section of the script generates the following XML when the user clicks the login button:

```
<login username="JeanSmith" password="VerySecret" />
```

The server receives the XML, generates an XML response and sends it back to the SWF file. If the password is accepted, the server responds with the following:

```
<loginreply status="OK" session="46597865D0B7884E007DEA7D" />
```

This XML includes a `session` attribute that contains a unique, randomly generated session ID, which will be used in all communications between the client and server for the rest of the session. If the password is rejected, the server responds with the following message:

```
<loginreply status="FAILURE" />
```

The `loginreply` XML node must load into a blank XML object in the SWF file. The following statement creates the XML object `loginreplyXML` to receive the XML node:

```
// Construct an XML object to hold the server's reply
var loginReplyXML:XML = new XML();
loginReplyXML.onLoad = function(success:Boolean) {
```

The second statement in this ActionScript defines an anonymous (inline) function, which is called when the `onLoad` event triggers.

The login button (`login_btn` instance) is used to send the user name and password as XML to the server and to load an XML response back into the SWF file. You can use the `sendAndLoad()` method to do this, as shown in the following example:

```
loginXML.sendAndLoad("http://www.flash-mx.com.com/mm/main.cfm",
  loginReplyXML);
```

First, the XML-formatted data is created, using the values that the user inputs in the SWF file, and that XML object is sent using the `sendAndLoad` method. Similar to data from a `loadVariables()` function, the `loginreply` XML element arrives asynchronously (that is, it doesn't wait for results before being returned) and loads into the `loginReplyXML` object. When the data arrives, the `onLoad` handler of the `loginReplyXML` object is called. You must define the `loginReplyXML` function, which is called when the `onLoad` handler triggers, so it can process the `loginreply` element.

Note: This function must always be on the frame that contains the ActionScript for the login button.

If the login is successful, the SWF file progresses to the `welcome` frame label. If the login is not successful, then the playhead moves to the `loginfailure` frame label. This is processed using a condition and case statement. For more information on case and break statements, see `case` and `break` in *Flash ActionScript Language Reference*. For more information on conditions, see `if` and `else` in *Flash ActionScript Language Reference*.

Note: This design is only an example, and Macromedia can make no claims about the level of security it provides. If you are implementing a secure password-protected system, make sure you have a good understanding of network security.

For more information, see Integrating XML and Flash in a Web Application at www.macromedia.com/support/flash/interactivity/xml/ and the "XML class" entry in *Flash ActionScript Language Reference*.

Using the XMLSocket class

ActionScript provides a built-in XMLSocket class, which lets you open a continuous connection with a server. A socket connection lets the server publish, or *push*, information to the client as soon as that information is available. Without a continuous connection, the server must wait for an HTTP request. This open connection removes latency issues and is commonly used for real-time applications such as chats. The data is sent over the socket connection as one string and should be formatted as XML. You can use the XML class to structure the data.

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.

Note: The XMLSocket class cannot tunnel through firewalls automatically because, unlike RTMP protocol, XMLSocket has no HTTP tunneling capability. If you need to use HTTP tunneling, consider using Flash Remoting or Flash Communication Server (which supports RTMP) instead.

You can use the `connect()` and `send()` methods of the XMLSocket class to transfer XML to and from a server over a socket connection. The `connect()` method establishes a socket connection with a web server port. The `send()` method passes an XML object to the server specified in the socket connection.

When you invoke the `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 `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).

The following example creates an XML socket connection and sends data from the XML object `myXML`. To understand the script, read the commented lines (indicated by the characters `//`):

```
//create XMLSocket object
var theSocket:XMLSocket = new XMLSocket();
//connect to a site on unused port above 1024 using connect() method
//enter localhost or 127.0.0.1 for local testing
//for live server enter your domain www.yourdomain.com
theSocket.connect("localhost", 12345);
//displays text regarding connection
theSocket.onConnect = function(myStatus) {
  if (myStatus) {
    conn_txt.text = "connection successful";
  } else {
    conn_txt.text = "no connection made";
  }
};
//data to send
function sendData() {
  var myXML:XML = new XML();
  var mySend = myXML.createElement("thenode");
  mySend.attributes.myData = "someData";
  myXML.appendChild(mySend);
```

```
  theSocket.send(myXML);
}
//button sends data
sendButton.onRelease = function() {
  sendData();
};
//traces data returned from socket connection
theSocket.onData = function(msg:String):Void {
  trace(msg);
};
```

For more information, see the "XMLSocket class" entry in *Flash ActionScript Language Reference*.

Sending messages to and from Flash Player

To send messages from a SWF file to its host environment (for example, a web browser, a Macromedia Director movie, or the stand-alone Flash Player), you can use the `fscommand()` function. This function lets you extend your SWF file by using the capabilities of the host. For example, you could pass an `fscommand()` function to a JavaScript function in an HTML page that opens a new browser window with specific properties.

To control a SWF file in Flash Player from web browser scripting languages such as JavaScript, VBScript, and Microsoft JScript, you can use Flash Player methods—functions that send messages from a host environment to the SWF file. For example, you could have a link in an HTML page that sends your SWF file to a specific frame.For more information, see the following topics:

- "Using fscommand()" on page 285
- "About Flash Player methods" on page 287
- "About using JavaScript to control Flash applications" on page 288

Using fscommand()

You use the `fscommand()` function to send a message to whichever program is hosting Flash Player, such as a web browser.

Note: Using `fscommand` to call Javascript does not work on the Safari or Internet Explorer browsers for the Macintosh.

The `fscommand()` function has two parameters: *`command`* and *`arguments`*. To send a message to the stand-alone version of Flash Player, you must use predefined commands and arguments. For example, the following event handler sets the stand-alone player to scale the SWF file to the full monitor screen size when the button is released:

```
my_btn.onRelease = function() {
  fscommand("fullscreen", true);
};
```

The following table shows the values you can specify for the *command* and *arguments* parameters of `fscommand()` to control the playback and appearance of a SWF file playing in the stand-alone player, including projectors.

Command	Arguments	Purpose
`quit`	None	Closes the projector.
`fullscreen`	`true` or `false`	Specifying `true` sets Flash Player to full-screen mode. Specifying `false` returns the player to normal menu view.
`allowscale`	`true` or `false`	Specifying `false` sets the player so that the SWF file is always drawn at its original size and never scaled. Specifying `true` forces the SWF file to scale to 100% of the player.
`showmenu`	`true` or `false`	Specifying `true` enables the full set of context menu items. Specifying `false` dims all the context menu items except Settings and About Flash Player.
`exec`	Path to application	Executes an application from within the projector.

To use `fscommand()` to send a message to a scripting language such as JavaScript in a web browser, you can pass any two parameters in the *command* and *arguments* parameters. These parameters can be strings or expressions and will be used in a JavaScript function that "catches," or handles, the `fscommand()` function.

An `fscommand()` function invokes the JavaScript function *moviename*`_DoFSCommand` in the HTML page that embeds the SWF file, where *moviename* is the name of Flash Player as assigned by the `name` attribute of the `embed` tag or the `id` attribute of the `object` tag. If Flash Player is assigned the name `myMovie`, the JavaScript function invoked is `myMovie_DoFSCommand`.

To use `fscommand()` to open a message box from a SWF file in the HTML page through JavaScript:

1. Create a new FLA file, and save it as myMovie.fla.
2. Drag two instances of the Button component to the Stage and give them the instance names `window_btn` and `alert_btn`, respectively, and the labels Open Window and Alert.
3. Insert a new layer on the Timeline, and rename it Actions.
4. Select Frame 1 of the Actions layer, and add the following ActionScript in the Actions panel:

```
window_btn.onRelease = function() {
  fscommand("popup", "http://www.macromedia.com/");
};
alert_btn.onRelease = function() {
  fscommand("alert", "You clicked the button.");
};
```

5. Select File > Publish Settings, and make sure that Flash with FSCommand is selected in the Template menu on the HTML tab.
6. Select File > Publish to generate the SWF and HTML files.

7. In an HTML or text editor, open the HTML file that was generated in step 6 and examine the code. When you published your SWF file using the Flash with FSCommand template on the HTML tab of the Publish Settings dialog box, some additional code was inserted in the HTML file. The SWF file's `NAME` and `ID` attributes are the filename. For example, for the file myMovie.fla, the attributes would be set to `myMovie`.
8. In the HTML file, add the following JavaScript code where the document says `// Place your code here.`:

```
if (command == "alert") {
alert(args);
}
else if (command == "popup") {
window.open(args, "mmwin", "width=500,height=300");
}
```

   Alternatively, for Microsoft Internet Explorer applications, you can attach an event handler directly in the `<SCRIPT>` tag, as shown in this example:

```
<Script Language = "JavaScript" event="FSCommand (command, args)" for=
  "theMovie">
...
</Script>
```

9. Save and close the HTML file.

   When you're editing HTML files outside of Flash in this way, remember that you must deselect the HTML check box in File > Publish Settings, or your HTML code will be overwritten by Flash when you republish.
10. In a web browser, open the HTML file to view it. Click the Open Window button; a window opens to the Macromedia website. Click the Alert button; an alert window opens.

The `fscommand()` function can send messages to Macromedia Director that are interpreted by Lingo as strings, events, or executable Lingo code. If the message is a string or an event, you must write the Lingo code to receive it from the `fscommand()` function and carry out an action in Director. For more information, see the Director Support Center at www.macromedia.com/support/director.

In Visual Basic, Visual C++, and other programs that can host ActiveX controls, `fscommand()` sends a VB event with two strings that can be handled in the environment's programming language. For more information, use the keywords *Flash method* to search the Flash Support Center at www.macromedia.com/support/flash.

About Flash Player methods

You can use Flash Player methods to control a SWF file in Flash Player from web-browser scripting languages such as JavaScript and VBScript. As with other methods, you can use Flash Player methods to send calls to SWF files from a scripting environment other than ActionScript. Each method has a name, and most methods take parameters. A parameter specifies a value upon which the method operates. The calculation performed by some methods returns a value that can be used by the scripting environment.

There are two technologies that enable communication between the browser and Flash Player: LiveConnect (Netscape Navigator 3.0 or later on Windows 95/98/2000/NT/XP or Power Macintosh) and ActiveX (Internet Explorer 3.0 and later on Windows 95/98/2000/NT/XP). Although the techniques for scripting are similar for all browsers and languages, there are additional properties and events available for use with ActiveX controls.

For more information, including a complete list of Flash Player scripting methods, use the keywords *Flash method* to search the Flash Support Center at www.macromedia.com/support/flash.

About using JavaScript to control Flash applications

Flash Player 6 (6.0.40.0) and later supports certain JavaScript methods that are specific to Flash applications, as well as `FSCommand`, in Netscape 6.2 and later. Earlier versions do not support these JavaScript methods and `FSCommand` in Netscape 6.2 or later. For more information, see the Macromedia Support Center article, "Scripting With Flash" at www.macromedia.com/support/flash/publishexport/scriptingwithflash/.

For Netscape 6.2 and later, you do not need to set the `swliveconnect` attribute to `true`. However, setting `swLiveConnect` to `true` has no adverse effects on your SWF file. For more information, see the `swLiveConnect` attribute in "Parameters and attributes" in *Using Flash*.

Flash Player security features

By default, Flash Player 7 and later prevents a SWF file served from one domain from accessing data, objects, or variables from SWF files that are served from different domains. In addition, content that is loaded through nonsecure (non-HTTPS) protocols cannot access content loaded through a secure (HTTPS) protocol, even when both are in exactly the same domain. For example, a SWF file located at http://www.macromedia.com/main.swf cannot load data from https://www.macromedia.com/data.txt without explicit permission; neither can a SWF file served from one domain load data (using `loadVars()`, for example) from another domain.

Identical numeric IP addresses are compatible. However, a domain name is not compatible with an IP address, even if the domain name resolves to the same IP address.

The following table shows examples of compatible domains:

www.macromedia.com	www.macromedia.com
data.macromedia.com	data.macromedia.com
65.57.83.12	65.57.83.12

The following table shows examples of incompatible domains:

www.macromedia.com	data.macromedia.com
macromedia.com	www.macromedia.com
www.macromedia.com	macromedia.com
65.57.83.12	www.macromedia.com (even if this domain resolves to 65.57.83.12)
www.macromedia.com	65.57.83.12 (even if www.macromedia.com resolves to this IP)

For more information, see the following topics:

- "About allowing data access between cross-domain SWF files" on page 289
- "About allowing HTTP to HTTPS protocol access between SWF files" on page 290
- "About allowing cross-domain data loading" on page 290
- "About custom policy file locations" on page 292
- "About XMLSocket policy files" on page 293
- "About compatibility with previous Flash Player security models" on page 294.

About allowing data access between cross-domain SWF files

One SWF file can load another SWF file from any location on the Internet. However, for the two SWF files to access each other's data (variables and objects), the two files must originate from the same domain. By default, in Flash Player 7 and later, the two domains must match exactly for the two files to share data. However, a SWF file can grant access to SWF files served from specific domains by calling `LocalConnection.allowDomain` or `System.security.allowDomain()`.

For example, suppose main.swf is served from www.macromedia.com. That SWF file then loads another SWF file (data.swf) from data.macromedia.com into a movie clip instance that's created dynamically using `createEmptyMovieClip`.

```
// In macromedia.swf
this.createEmptyMovieClip("target_mc", this.getNextHighestDepth());
target_mc.loadMovie("http://data.macromedia.com/data.swf");
```

Now suppose that data.swf defines a method named `getData()` on its main Timeline. By default, main.swf cannot call the `getData()` method defined in data.swf after that file has loaded because the two SWF files do not reside in the same domain. For example, the following method call in main.swf, after data.swf has loaded, will fail:

```
// In macromedia.swf, after data.swf has loaded:
target_mc.getData(); // This method call will fail
```

However, data.swf can grant access to SWF files served from www.macromedia.com by using the `LocalConnection.allowDomain` handler and the `System.security.allowDomain()` method, depending on the type of access required. The following code, added to data.swf, allows a SWF file served from www.macromedia.com to access its variables and methods:

```
// Within data.swf
this._lockroot = true;
System.security.allowDomain("www.macromedia.com");
var my_lc:LocalConnection = new LocalConnection();
my_lc.allowDomain = function(sendingDomain) {
  return (sendingDomain == "www.macromedia.com");
};
function getData() {
  var timestamp:Date = new Date();
  output_txt.text += "data.swf:"+timestamp.toString()+"\n\n";
}
output_txt.text = "**INIT**:\n\n";
```

Now the `getData` function in the loaded SWF file can be called by the macromedia.swf file. Notice that `allowDomain` permits any SWF file in the allowed domain to script any other SWF file in the domain permitting the access, unless the SWF file being accessed is hosted on a site using a secure protocol (HTTPS). In this case, you must use `allowInsecureDomain` instead of `allowDomain`; see the following section.

For more information on domain-name matching, see "Flash Player security features" on page 288.

About allowing HTTP to HTTPS protocol access between SWF files

As discussed in the previous section, you must use an `allowDomain` handler or method to permit a SWF file in one domain to be accessed by a SWF file in another domain.However, if the SWF file being accessed is hosted at a site that uses a secure protocol (HTTPS), the `allowDomain` handler or method doesn't permit access from a SWF file hosted at a site that uses an insecure protocol. To permit such access, you must use the `LocalConnection.allowInsecure Domain()` or `System.security.allowInsecureDomain()` statements.

For example, if the SWF file at https://www.someSite.com/data.swf must allow access by a SWF file at http://www.someSite.com, the following code added to data.swf allows this access:

```
// Within data.swf
System.security.allowInsecureDomain("www.someSite.com");
my_lc.allowInsecureDomain = function(sendingDomain) {
  return(sendingDomain=="www.someSite.com");
}
```

About allowing cross-domain data loading

A Flash document can load data from an external source by using one of the following data loading calls: `XML.load()`, `XML.sendAndLoad()`, `LoadVars.load()`, `LoadVars.sendAndLoad()`, `loadVariables()`, `loadVariablesNum()`. Also, a SWF file can import runtime shared libraries, or assets defined in another SWF file, at runtime. By default, the data or SWF media, in the case of runtime shared libraries, must reside in the same domain as the SWF file that is loading that external data or media.

To make data and assets in runtime shared libraries available to SWF files in different domains, use a *cross-domain policy file*. A cross-domain policy file is an XML file that provides a way for the server to indicate that its data and documents are available to SWF files served from certain domains or from all domains. Any SWF file that is served from a domain specified by the server's policy file will be permitted to access data or assets from that server.

When a Flash document attempts to access data from another domain, Flash Player automatically attempts to load a policy file from that domain. If the domain of the Flash document that is attempting to access the data is included in the policy file, the data is automatically accessible.

Policy files must be named crossdomain.xml, and can reside either at the root directory or in another directory on the server that is serving the data with some additional ActionScript (see "About custom policy file locations" on page 292). Policy files function only on servers that communicate over HTTP, HTTPS, or FTP. The policy file is specific to the port and protocol of the server where it resides.

For example, a policy file located at https://www.macromedia.com:8080/crossdomain.xml will apply only to data loading calls made to www.macromedia.com over HTTPS at port 8080.

An exception to this rule is the use of an XMLSocket object to connect to a socket server in another domain. In that case, an HTTP server running on port 80 in the same domain as the socket server must provide the policy file for the method call.

An XML policy file contains a single `<cross-domain-policy>` tag, which, in turn, contains zero or more `<allow-access-from>` tags. Each `<allow-access-from>` tag contains an attribute, `domain`, which specifies either an exact IP address, an exact domain, or a wildcard domain (any domain). Wildcard domains are indicated by either a single asterisk (*), which matches all domains and all IP addresses, or an asterisk followed by a suffix, which matches only those domains that end with the specified suffix. Suffixes must begin with a dot. However, wildcard domains with suffixes can match domains that consist of only the suffix without the leading dot. For example, foo.com is considered to be part of *.foo.com. Wildcards are not allowed in IP domain specifications.

If you specify an IP address, access is granted only to SWF files loaded from that IP address using IP syntax (for example, http://65.57.83.12/flashmovie.swf), not those loaded using domain-name syntax. Flash Player does not perform DNS resolution.

The following example shows a policy file that permits access to Flash documents that originate from foo.com, www.friendOfFoo.com, *.foo.com, and 105.216.0.40, from a Flash document on foo.com:

```
<?xml version="1.0"?>
<!-- http://www.foo.com/crossdomain.xml -->
<cross-domain-policy>
  <allow-access-from domain="www.friendOfFoo.com" />
  <allow-access-from domain="*.foo.com" />
  <allow-access-from domain="105.216.0.40" />
</cross-domain-policy>
```

You can also permit access to documents originating from any domain, as shown in the following example:

```
<?xml version="1.0"?>
<!-- http://www.foo.com/crossdomain.xml -->
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

Each `<allow-access-from>` tag also has the optional `secure` attribute. The `secure` attribute defaults to `true`. You can set the attribute to `false` if your policy file is on an HTTPS server, and you want to allow SWF files on an HTTP server to load data from the HTTPS server.

Setting the `secure` attribute to `false` could compromise the security offered by HTTPS.

If the SWF file you are downloading comes from a HTTPS server, but the SWF file loading it is on an HTTP server, you need to add the `secure="false"` attribute to the `<allow-access-from>` tag, as shown in the following code:

```
<allow-access-from domain="www.foo.com" secure="false" />
```

A policy file that contains no `<allow-access-from>` tags has the same effect as not having a policy on a server.

About custom policy file locations

Flash Player 7 (7.0.19.0) supports a new method called `System.security.loadPolicyFile`. This method lets you specify a custom location on a server where a cross domain policy file can be found, so it does not need to be in the root directory. Flash Player 7 (7.0.14.0) only searched for policy files in the root location of a server, but it can be inconvenient for a site administrator to place this file in the root directory. For more information on the `loadPolicyFile` method and XMLSocket connections, see "About XMLSocket policy files" on page 293 and `System.security.loadPolicyFile` in *Flash ActionScript Language Reference*.

If you use the `loadPolicyFile` method, a site administrator can place the policy file in any directory, as long as the SWF files that need to use the policy file call `loadPolicyFile` to tell Flash Player where the policy file is located. However, policy files not placed in the root directory have a limited scope. The policy file only allows access to locations at or below its own level in the server's hierarchy.

The `loadPolicyFile` method is available only in Flash Player 7 (7.0.19.0) or greater. Authors of SWF files using the `loadPolicyFile` method must do one of the following:

- Require Flash Player 7 (7.0.19.0) or later.
- Arrange for the site where the data is coming from to have a policy file in the default location (the root directory) as well as in the non-default location. Earlier versions of Flash Player will use the default location.

Otherwise, authors must create SWF files so a failure of a cross-domain loading operation is implemented.

Caution: If your SWF file relies on loadPolicyFile, visitors with Flash Player 6 or earlier or Flash Player 7 (7.0.19.0) or later will not have problems. However, visitors with Flash Player 7 (7.0.14.0) will not have support for loadPolicyFile.

If you want to use a policy file in a custom location on the server, you must call `System.security.loadPolicyFile` *before* you make any requests that depend on the policy file, such as the following:

```
System.security.loadPolicyFile
    ("http://www.foo.com/folder1/folder2/crossdomain.xml");
var my_xml:XML = new XML();
my_xml.load("http://www.foo.com/folder1/folder2/myData.xml");
```

You can load several policy files with overlapping scopes using `loadPolicyFile`. For all requests, Flash Player tries to consult all the files whose scope includes the location of the request. If one policy file fails to grant cross domain access, another file is not prevented from granting access to data. If all access attempts fail, Flash Player looks in the default location of the crossdomain.xml file (in the in the root directory). The request fails if no policy file is found in the default location.

About XMLSocket policy files

For an XMLSocket connection attempt, Flash Player 7 (7.0.14.0) looked for crossdomain.xml on an HTTP server on port 80 in the subdomain to which the connection attempt was being made. Flash Player 7 (7.0.14.0) and all earlier versions restricted XMLSocket connections to ports 1024 and above. However, in Flash Player 7 (7.0.19.0) and later, ActionScript can inform Flash Player of a non-default location for a policy file using `System.security.loadPolicyFile`. Any custom locations for XMLSocket policy files must still be on an XML socket server.

In the following example, Flash Player retrieves a policy file from a specified URL:

```
System.security.loadPolicyFile("http://www.foo.com/folder/policy.xml");
```

Any permissions granted by the policy file at that location apply to all content at the same level or below in the server's hierarchy. Therefore, if you try to load the following data, you discover you can only load data from certain locations:

```
myLoadVars.load("http://foo.com/sub/dir/vars.txt"); // allowed
myLoadVars.load("http://foo.com/sub/dir/deep/vars2.txt"); // allowed
myLoadVars.load("http://foo.com/elsewhere/vars3.txt"); // not allowed
```

To work around this, you can load more than one policy file into a single SWF file using `loadPolicyFile`. Flash Player always waits for the completion of any policy file downloads before denying a request that requires a policy file. Flash Player consults the default location of crossdomain.xml if no other policies were authorized in the SWF file.

New syntax allows policy files to be retrieved directly from an XMLSocket server:

```
System.security.loadPolicyFile("xmlsocket://foo.com:414");
```

In this example, Flash Player tries to retrieve a policy file from the specified host and a port. Any port can be used if the policy file is not in the default (root) directory; otherwise the port is limited to 1024 and higher (as with earlier players). When a connection is established to the specified port, Flash Player sends `<cross-domain-request/>`, terminated by a null byte.

The XML socket server might be configured to serve policy files in the following ways:

- To serve policy files and normal socket connections over the same port. The server should wait for `<cross-domain-request />` before transmitting a policy file.
- To serve policy files over a separate port from normal connections, in which case it might send a policy file as soon as a connection is established on the dedicated policy file port.

The server must send a null byte to terminate a policy file before it closes the connection. If the server does not close the connection, Flash Player will do so upon receiving the terminating null byte.

A policy file served by an XML socket server has the same syntax as any other policy file, except that it must also specify the ports to which access is granted. The allowed ports are specified in a `to-ports` attribute in the `<allow-access-from>` tag. If a policy file is less than port 1024, it can grant access to any port; when a policy file comes from port 1024 or higher, it can grant access only to other ports above 1024. Single port numbers, port ranges and wildcards are allowed. The following code is an example of a XMLSocket policy file:

```
<cross-domain-policy>
<allow-access-from domain="*" to-ports="507" />
```

```
<allow-access-from domain="*.foo.com" to-ports="507,516" />
<allow-access-from domain="*.bar.com" to-ports="516-523" />
<allow-access-from domain="www.foo.com" to-ports="507,516-523" />
<allow-access-from domain="www.bar.com" to-ports="*" />
</cross-domain-policy>
```

Because the ability to connect to ports lower than 1024 is new in Flash Player 7 (7.0.19.0), a policy file loaded with `loadPolicyFile` is always required to authorize this, even when a SWF file is connecting to its own subdomain.

About compatibility with previous Flash Player security models

As a result of the security feature changes in Flash Player (see "Flash Player security features" on page 288), content that runs properly in Flash Player 6 or earlier might not run properly in Flash Player 7 or later.

For example, in Flash Player 6, a SWF file that resides in www.macromedia.com could access data on a server located at data.macromedia.com. That is, Flash Player 6 allowed a SWF file from one domain to load data from a "similar" domain.

In Flash Player 7 and later, if a version 6 (or earlier) SWF file attempts to load data from a server that resides in another domain, and that server doesn't provide a policy file that allows access from that SWF file's domain, then the Macromedia Flash Player Settings dialog box appears. The dialog box asks the user to allow or deny the cross-domain data access.

If the user clicks Allow, the SWF file is permitted to access the requested data; if the user clicks Deny, the SWF file is not allowed to access the requested data.

To prevent this dialog box from appearing, create a security policy file on the server providing the data. For more information, see "About allowing cross-domain data loading" on page 290.

CHAPTER 12
Working with External Media

If you import an image or a sound while you author a document in Macromedia Flash MX 2004 or Macromedia Flash MX Professional 2004, the image and sound are packaged and stored in the SWF file when you publish it. In addition to importing media while authoring, you can load external media, including other SWF files, at runtime. There are several reasons you might want to keep media outside a Flash document.

Reduce file size By keeping large media files outside your Flash document and loading them at runtime, you can reduce the initial download time for your applications and presentations, especially over slow Internet connections.

Modularize large presentations You can divide a large presentation or application into separate SWF files and then load those separate files as needed at runtime. This process reduces initial download time and also makes maintaining and updating the contents of the presentation easier.

Separate content from presentation This theme is common in application development, especially data-driven applications. For example, a shopping cart application might display a JPEG image of each product. By loading the JPEG files for each image at runtime, you can easily update a product's image without modifying the original FLA file.

Take advantage of runtime-only features Some features, such as dynamically loaded FLV and MP3 playback, are available only at runtime through ActionScript.

Overview of loading external media

There are four types of media files that you can load into a Flash application at runtime: SWF, MP3, JPEG, and FLV files. Macromedia Flash Player can load external media from any HTTP or FTP address, from a local disk using a relative path, or by using the `file://` protocol.

To load external SWF and JPEG files, you can use the `loadMovie()` or `loadMovieNum()` function, the `MovieClip.loadMovie()` method, or the `MovieClipLoader.loadClip()` method. The class methods generally provide more function and flexibility than global functions and are appropriate for more complex applications. When you load a SWF or JPEG file, you specify a movie clip or movie level as the target for that media. For more information on loading SWF and JPEG files, see "Loading external SWF and JPEG files" on page 296.

To play back an external MPEG Layer 3 (MP3) file, use the `loadSound()` method of the Sound class. This method lets you specify whether the MP3 file should progressively download or finish downloading completely before it starts to play. You can also read the ID3 information embedded in MP3 files, if they're available. For more information, see "Reading ID3 tags in MP3 files" on page 298.

Flash Video (FLV) is the native video format used by Flash Player. You can play back FLV files over HTTP or from the local file system. Playing external FLV files provides several advantages over embedding video in a Flash document, such as better performance and memory management, and independent video and Flash frame rates. For more information, see "Playing back external FLV files dynamically" on page 299.

You can also preload or track the download progress of external media. Flash Player 7 introduces the MovieClipLoader class, which you can use to track the download progress of SWF or JPEG files. To preload MP3 and FLV files, you can use the `getBytesLoaded()` method of the Sound class and the `bytesLoaded` property of the NetStream class. For more information, see "Preloading external media" on page 300.

Loading external SWF and JPEG files

To load a SWF or JPEG file, use the `loadMovie()` or `loadMovieNum()` global function, the `loadMovie()` method of the MovieClip class, or the `loadClip()` method of the MovieClipLoader class. For more information on the `loadClip()` method, see `MovieClipLoader.loadClip()` in *Flash ActionScript Language Reference*

To load a SWF or JPEG file into a level in Flash Player, use `loadMovieNum()`. To load a SWF or JPEG file into a movie clip target, use the `loadMovie()` function or method. In either case, the loaded content replaces the content of the specified level or target movie clip.

When you load a SWF or JPEG file into a movie clip target, the upper left corner of the SWF file or JPEG image is placed on the registration point of the movie clip. Because this registration point is often the center of the movie clip, the loaded content might not appear centered. Also, when you load a SWF file or JPEG image to a root Timeline, the upper left corner of the image is placed on the upper left corner of the Stage. The loaded content inherits rotation and scaling from the movie clip, but the original content of the movie clip is removed.

You can optionally send ActionScript variables with a `loadMovie()` or `loadMovieNum()` call. This is useful, for example, if the URL you're specifying in the method call is a server-side script that returns a JPEG or SWF file according to data passed from the Flash application.

For image files, Flash supports only the standard JPEG image file type, not progressive JPEG files.

When you use the global `loadMovie()` or `loadMovieNum()` function, specify the target level or clip as a parameter. For example, the following code loads the Flash application contents.swf into the movie clip instance named `target_mc`:

```
loadMovie("contents.swf", target_mc);
```

You can use `MovieClip.loadMovie()` to achieve the same result:

```
target_mc.loadMovie("contents.swf");
```

The following code loads the JPEG image flowers.jpg into the movie clip instance `image_clip`:

```
image_clip.loadMovie("flowers.jpg");
```

For more information about these methods, see `loadMovie()`, `loadMovieNum()`, and `MovieClip.loadMovie()` in *Flash ActionScript Language Reference*.

For more information about loading external SWF and JPEG files, see the next section.

About loaded SWF files and the root Timeline

The ActionScript property `_root` specifies or returns a reference to the root Timeline of a SWF file. If you load a SWF file into a movie clip in another SWF file, any references to `_root` in the loaded SWF file resolve to the root Timeline in the host SWF file, not to that of the loaded SWF file. This can sometimes cause unexpected behavior at runtime (for example, if the host SWF file and the loaded SWF file both use `_root` to specify a variable).

In Flash Player 7 and later, you can use the `MovieClip._lockroot` property to force references to `_root` made by a movie clip to resolve to its own Timeline rather than to the Timeline of the SWF file that contains that movie clip. For more information, see "Specifying a root Timeline for loaded SWF files" on page 208. For more information about using `_root` and `_lockroot`, see Chapter 3, "Using scope," on page 95.

One SWF file can load another SWF file from any location on the Internet. However, for one SWF file to access data (variables, methods, and so forth) defined in the other SWF file, the two files must originate from the same domain. In Flash Player 7 and later, cross-domain scripting is prohibited unless the loaded SWF file specifies otherwise by calling `System.security.allowDomain()`.

For more information, see `System.security.allowDomain()` in *Flash ActionScript Language Reference* and "Flash Player security features" on page 288.

Loading external MP3 files

To load MP3 files at runtime, you use the `loadSound()` method of the Sound class. First, you create a Sound object, as shown in the following example:

```
var song1_sound:Sound = new Sound();
```

Then you use the new object to call `loadSound()` to load an event or a streaming sound. Event sounds are loaded completely before being played; streaming sounds play as they are downloaded. You can set the *`isStreaming`* parameter of `loadSound()` to specify a sound as an event sound or a streaming sound. After you load an event sound, you must call the `start()` method of the Sound class to make the sound play. Streaming sounds begin playing when sufficient data is loaded into the SWF file; you don't need to use `start()`.

For example, the following code creates a Sound object named `classical` and then loads an MP3 file named beethoven.mp3:

```
var classical:Sound = new Sound();
classical.loadSound("http://server.com/mp3s/beethoven.mp3", true);
```

In most cases, set the *isStreaming* parameter to `true`, especially if you're loading large sound files that should start playing as soon as possible—for example, when creating an MP3 "jukebox" application. However, if you're downloading shorter sound clips and need to play them at a specified time (for example, when a user clicks a button), set *isStreaming* to `false`.

To determine when a sound has completely downloaded, use the `Sound.onLoad` event handler. This event handler automatically receives a Boolean (`true` or `false`) value that indicates whether the file downloaded successfully.

For example, suppose you're creating an online game that uses different sounds depending on what level the user has reached in the game. The following code loads an MP3 file (blastoff.mp3) into a Sound object named `gameSound`, and then plays the sound when it has completely downloaded:

```
var gameSound:Sound = new Sound();
gameSound.onLoad = function (loadedOK) {
  if(loadedOK) {
    gameSound.start();
  }
}
gameSound.loadSound("http://server.com/sounds/blastoff.mp3", false);
```

For sound files, Flash Player supports only the MP3 sound file type.

For more information, see `Sound.loadSound()`, `Sound.start()`, and `Sound.onLoad` in *Flash ActionScript Language Reference*.

Reading ID3 tags in MP3 files

ID3 tags are data fields added to an MP3 file that contain information about the file, such as the song name, album name, and artist name.

To read ID3 tags from an MP3 file, use the `Sound.id3` property, whose properties correspond to the names of ID3 tags included in the MP3 file being loaded. To determine when ID3 tags for a downloading MP3 file are available, use the `Sound.onID3` event handler. Flash Player 7 supports version 1.0, 1.1, 2.3, and 2.4 tags; version 2.2 tags are not supported.

For example, the following code loads an MP3 file named favoriteSong.mp3 into the Sound object named `song`. When the ID3 tags for the file are available, a text field named `display_txt` shows the artist name and song name.

```
var song:Sound = new Sound();
song.onID3 = function () {
  display_txt.text = "Artist: " + song.id3.TPE1 + newline;
  display_txt.text += "Song: " + song.id3.TIT2);
}
song.loadSound("mp3s/favoriteSong.mp3", true);
```

Because ID3 2.0 tags are located at the beginning of an MP3 file (before the sound data), they are available as soon as the file starts downloading. ID3 1.0 tags, however, are located at the end of the file (after the sound data), so they aren't available until the entire MP3 file finishes downloading.

The `onID3` event handler is called each time new ID3 data is available. This means that if an MP3 file contains ID3 2.0 tags and ID3 1.0 tags, the `onID3` handler will be called twice because the tags are located in different parts of the file.

For a list of supported ID3 tags, see `Sound.id3` in *Flash ActionScript Language Reference*.

Playing back external FLV files dynamically

As an alternative to importing video into the Flash authoring environment, you can use ActionScript to dynamically play back external FLV files in Flash Player. You can play back FLV files from an HTTP address or from the local file system. To play back FLV files, you use the NetConnection and NetStream classes and the `attachVideo()` method of the Video class. (For more information, see "NetConnection class", "NetStream class", and `Video.attachVideo()` in *Flash ActionScript Language Reference*.)

You can create FLV files by importing video into the Flash authoring tool and exporting it as an FLV file. (See "Macromedia Flash Video (FLV)" in *Using Flash*.) If you have Flash Professional, you can use the FLV Export plug-in to export FLV files from supported video-editing applications. (See "Exporting FLV files from video-editing applications (Flash Professional only)" in *Using Flash*.)

Using external FLV files provides certain capabilities that are not available when you use imported video:

- Longer video clips can be used in your Flash documents without slowing down playback. External FLV files are played using *cached memory*. This means that large files are stored in small pieces and accessed dynamically, which does not require as much memory as embedded video files.
- An external FLV file can have a different frame rate than the Flash document in which it plays. For example, you can set the Flash document frame rate to 30 fps and the video frame rate to 21 fps. This gives you greater control in ensuring smooth video playback.
- With external FLV files, Flash document playback does not have to be interrupted while the video file is loading. Imported video files can sometimes interrupt document playback to perform certain functions (for example, accessing a CD-ROM drive). FLV files can perform functions independently of the Flash document, which does not interrupt playback.
- Captioning of video content is easier with external FLV files because you can use event handlers to access metadata for the video.

Tip: To load FLV files from a web server, you might need to register the file extension and MIME type with your web server; check your web server documentation. The MIME type for FLV files is video/x-flv.

The following procedure shows how to play back a file named videoFile.flv that is stored in the same location as your SWF file.

To play back an external FLV file in a Flash document:

1. With the document open in the Flash authoring tool, in the Library panel (Window > Library) select New Video from the Library options menu to create a video object.
2. Drag a video object from the Library panel onto the Stage; this creates a video object instance.
3. With the video object selected on the Stage, in the Property inspector (Window > Properties) enter `my_video` in the Instance Name text box.
4. Open the Components panel (Window > Development Panels > Components), and drag a TextArea component to the Stage.
5. With the TextArea object selected on the Stage, enter `status` in the Instance Name text box in the Property inspector.
6. Select Frame 1 in the Timeline, and open the Actions panel (Window > Development Panels > Actions).
7. Add the following code to the Actions panel:

```
// Create a NetConnection object
var netConn:NetConnection = new NetConnection();
// Create a local streaming connection
netConn.connect(null);
// Create a NetStream object and define an onStatus() function
var netStream:NetStream = new NetStream(netConn);
netStream.onStatus = function(infoObject) {
  status_txt.text += "Status (NetStream)" + newline;
  status_txt.text += "Level: "+infoObject.level + newline;
  status_txt.text += "Code: "+infoObject.code + newline;
};
// Attach the NetStream video feed to the Video object
my_video.attachVideo(netStream);
// Set the buffer time
netStream.setBufferTime(5);
// Begin playing the FLV file
netStream.play("videoFile.flv");
```

Preloading external media

ActionScript provides several ways to preload or track the download progress of external media. To preload SWF and JPEG files, use the MovieClipLoader class, which provides an event listener mechanism for checking download progress. This class is new in Flash Player 7. For more information, see "Preloading SWF and JPEG files" on page 301.

To track the download progress of MP3 files, use the `Sound.getBytesLoaded()` and `Sound.getBytesTotal()` methods; to track the download progress of FLV files, use the `NetStream.bytesLoaded` and `NetStream.bytesTotal` properties. For more information, see "Preloading MP3 and FLV files" on page 302.

Preloading SWF and JPEG files

To preload SWF and JPEG files into movie clip instances, you can use the "MovieClipLoader class". This class provides an event listener mechanism to give notification about the status of file downloads into movie clips. Using a MovieClipLoader object to preload SWF and JPEG files involves the following steps:

Create a new MovieClipLoader object You can use a single MovieClipLoader object to track the download progress of multiple files, or create a separate object for each file's progress. Create a new movie clip, load your contents into it, and then create the MovieClipLoader object.

```
this.createEmptyMovieClip("target_mc", 999);
var loader:MovieClipLoader = new MovieClipLoader();
```

Create a listener object and create event handlers The listener object can be any ActionScript object, such as a generic Object object, a movie clip, or a custom component.

For example, the following code creates a generic listener object named `loadListener` and defines for itself `onLoadStart`, `onLoadProgress`, and `onLoadComplete` functions:

```
var loader:MovieClipLoader = new MovieClipLoader();
// Create listener object:
var loadListener:Object = new Object();
loadListener.onLoadStart = function(loadTarget) {
  trace("Loading into "+loadTarget+" has started.");
};
loadListener.onLoadProgress = function(loadTarget, bytesLoaded, bytesTotal) {
  var percentLoaded = bytesLoaded/bytesTotal*100;
  trace("%"+percentLoaded+" into target "+loadTarget);
};
loadListener.onLoadComplete = function(loadTarget) {
  trace("Load completed into: "+loadTarget);
};
```

Register the listener object with the MovieClipLoader object In order for the listener object to receive the loading events, you must register it with the MovieClipLoader object, as shown in the following code:

```
loader.addListener(loadListener);
```

Begin loading the file (JPEG or SWF) into a target clip To start the download of the JPEG or SWF file, you use the `MovieClipLoader.loadClip()` method, as shown in the following code:

```
loader.loadClip("mymovie.swf", target_mc);
```

Note: You can use MovieClipLoader methods only to track the download progress of files loaded with the `MovieClipLoader.loadClip()` method. You cannot use the `loadMovie()` function or `MovieClip.loadMovie()` method.

The following example uses the `setProgress()` method of the ProgressBar component to display the download progress of a SWF file. (See "ProgressBar component" in *Using Components*.)

To display download progress using the ProgressBar component:

1. In a new Flash document, create a movie clip on the Stage and give it an instance name `target_mc`.
2. Open the Components panel (Window > Development Panels > Components).
3. Drag a ProgressBar component from the Components panel to the Stage.
4. In the Property inspector, give the ProgressBar component the name `pBar` and, on the Parameters tab, select Manual from the Mode pop-up menu.
5. Select Frame 1 in the Timeline, and open the Actions panel (Window > Development Panels > Actions).
6. Add the following code to the Actions panel:

```
// create both a MovieClipLoader object and a listener object
myLoader = new MovieClipLoader();
myListener = new Object();
// add the MovieClipLoader callbacks to your listener object
myListener.onLoadStart = function(clip) {
  // this event is triggered once, when the load starts
  pBar.label = "Now loading: " + clip;
};
myListener.onLoadProgress = function(clip, bytesLoaded, bytesTotal) {
  var percentLoaded = int (100*(bytesLoaded/bytesTotal));
  pBar.setProgress(bytesLoaded, bytesTotal);
};
myLoader.addListener(myListener);
myLoader.loadClip("veryLargeFile.swf", target_mc);
```

7. Test the document by selecting Control > Test Movie. You can see the movie load.
8. Publish to HTML, and open the HTML file in a browser to see the progress bar in action.

For more information, see "MovieClipLoader class" in *Flash ActionScript Language Reference*.

Preloading MP3 and FLV files

To preload MP3 and FLV files, you can use the `setInterval()` function to create a *polling* mechanism that checks the bytes loaded for a Sound or NetStream object at predetermined intervals. To track the download progress of MP3 files, use the `Sound.getBytesLoaded()` and `Sound.getBytesTotal()` methods; to track the download progress of FLV files, use the `NetStream.bytesLoaded` and `NetStream.bytesTotal` properties.

The following code uses `setInterval()` to check the bytes loaded for a Sound or NetStream object at predetermined intervals:

```
// Create a new Sound object to play the sound.
var songTrack:Sound = new Sound();
// Create the polling function that tracks download progress.
// This is the function that is "polled." It checks
// the download progress of the Sound object passed as a reference.
checkProgress = function (soundObj) {
  var bytesLoaded = soundObj.getBytesLoaded();
  var bytesTotal = soundObj.getBytesTotal();
  var percentLoaded = Math.floor(bytesLoaded/bytesTotal * 100);
```

```
  trace("%" + percentLoaded + " loaded.");
};
// When the file has finished loading, clear the interval polling.
songTrack.onLoad = function () {
  clearInterval(poll);
};
// Load streaming MP3 file and start calling checkProgress()
songTrack.loadSound("http://yourserver.com/songs/beethoven.mp3", true);
var poll = setInterval(checkProgress, 1000, songTrack);
```

You can use this same kind of polling technique to preload external FLV files. To get the total bytes and current number of bytes loaded for an FLV file, use the `NetStream.bytesLoaded` and `NetStream.bytesTotal` properties. Try loading your song from a server to see the loading progress in the Output panel.

Another way to preload FLV files is to use the `NetStream.setBufferTime()` method. This method takes a single parameter that indicates the number of seconds of the FLV stream to download before playback begins.

For more information, see `MovieClip.getBytesLoaded()`, `MovieClip.getBytesTotal()`, `NetStream.bytesLoaded`, `NetStream.bytesTotal`, `NetStream.setBufferTime()`, `setInterval()`, `Sound.getBytesLoaded()`, and `Sound.getBytesTotal()` in *Flash ActionScript Language Reference*.

APPENDIX A
Error Messages

Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004 provide enhanced compile-time error reporting when you publish to ActionScript 2.0 (the default). The following table contains a list of error messages that the Flash compiler can generate:

Error number	Message text
1093	A class name was expected.
1094	A base class name is expected after the 'extends' keyword.
1095	A member attribute was used incorrectly.
1096	The same member name may not be repeated more than once.
1097	All member functions need to have names.
1099	This statement is not permitted in a class definition.
1100	A class or interface has already been defined with this name.
1101	Type mismatch.
1102	There is no class with the name '<ClassName>'.
1103	There is no property with the name '<propertyName>'.
1104	A function call on a non-function was attempted.
1105	Type mismatch in assignment statement: found [lhs-type] where [rhs-type] is required.
1106	The member is private and cannot be accessed.
1107	Variable declarations are not permitted in interfaces.
1108	Event declarations are not permitted in interfaces.
1109	Getter/setter declarations are not permitted in interfaces.
1110	Private members are not permitted in interfaces.
1111	Function bodies are not permitted in interfaces.
1112	A class may not extend itself.
1113	An interface may not extend itself.

Error number	Message text
1114	There is no interface defined with this name.
1115	A class may not extend an interface.
1116	An interface may not extend a class.
1117	An interface name is expected after the 'implements' keyword.
1118	A class may not implement a class, only interfaces.
1119	The class must implement method 'methodName' from interface 'interfaceName'.
1120	The implementation of an interface method must be a method, not a property.
1121	A class may not extend the same interface more than once.
1122	The implementation of the interface method doesn't match its definition.
1123	This construct is only available in ActionScript 1.
1124	This construct is only available in ActionScript 2.0.
1125	Static members are not permitted in interfaces.
1126	The expression returned must match the function's return type.
1127	A return statement is required in this function.
1128	Attribute used outside class.
1129	A function with return type Void may not return a value.
1130	The 'extends' clause must appear before the 'implements' clause.
1131	A type identifier is expected after the ':'.
1132	Interfaces must use the 'extends' keyword, not 'implements'.
1133	A class may not extend more than one class.
1134	An interface may not extend more than one interface.
1135	There is no method with the name '<methodName>'.
1136	This statement is not permitted in an interface definition.
1137	A set function requires exactly one parameter.
1138	A get function requires no parameters.
1139	Classes may only be defined in external ActionScript 2.0 class scripts.
1140	ActionScript 2.0 class scripts may only define class or interface constructs.
1141	The name of this class, '<A.B.C>', conflicts with the name of another class that was loaded, '<A.B>'. (This error occurs when the ActionScript 2.0 compiler cannot compile a class because of the full name of an existing class is part of the conflicting class' name. For example, compiling class `mx.com.util` generates error 1141 if class `mx.com` is a compiled class.)
1142	The class '<ClassName>' could not be loaded.
1143	Interfaces may only be defined in external ActionScript 2.0 class scripts.

Error number	Message text
1144	Instance variables cannot be accessed in static functions.
1145	Class and interface definitions cannot be nested.
1146	The property being referenced does not have the static attribute.
1147	This call to super does not match the superconstructor.
1148	Only the public attribute is allowed for interface methods.
1149	The import keyword cannot be used as a directive.
1150	You must export your movie as Flash 7 to use this action.
1151	You must export your movie as Flash 7 to use this expression.
1152	This exception clause is placed improperly.
1153	A class must have only one constructor.
1154	A constructor may not return a value.
1155	A constructor may not specify a return type.
1156	A variable may not be of type Void.
1157	A function parameter may not be of type Void.
1158	Static members can only be accessed directly through classes.
1159	Multiple implemented interfaces contain same method with different types.
1160	There is already a class or interface defined with this name.
1161	Classes, interfaces, and built-in types may not be deleted.
1162	There is no class with this name.
1163	The keyword '‹keyword›' is reserved for ActionScript 2.0 and cannot be used here.
1164	Custom attribute definition was not terminated.
1165	Only one class or interface can be defined per ActionScript 2.0 as file.
1166	The class being compiled, '‹A.b›', does not match the class that was imported, '‹A.B›'. (This error occurs when a class name is spelled with a different case from an imported class. For example, compiling class `mx.com.util` generates error 1166 if the statement `import mx.Com` appears in the util.as file.)
1167	You must enter a class name.
1168	The class name you have entered contains a syntax error.
1169	The interface name you have entered contains a syntax error.
1170	The base class name you have entered contains a syntax error.
1171	The base interface name you have entered contains a syntax error.
1172	You must enter an interface name.
1173	You must enter a class or interface name.

Error number	Message text
1174	The class or interface name you have entered contains a syntax error.
1175	'variable' is not accessible from this scope.
1176	Multiple occurrences of the 'get/set/private/public/static' attribute were found.
1177	A class attribute was used incorrectly.
1178	Instance variables and functions may not be used to initialize static variables.
1179	Runtime circularities were discovered between the following classes: ‹list of user-defined classes›. This runtime error indicates that your custom classes are incorrectly referencing each other.
1180	The currently targeted Flash Player does not support debugging.
1181	The currently targeted Flash Player does not support the releaseOutside event.
1182	The currently targeted Flash Player does not support the dragOver event.
1183	The currently targeted Flash Player does not support the dragOut event.
1184	The currently targeted Flash Player does not support dragging actions.
1185	The currently targeted Flash Player does not support the loadMovie action.
1186	The currently targeted Flash Player does not support the getURL action.
1187	The currently targeted Flash Player does not support the FSCommand action.
1188	Import statements are not allowed inside class or interface definitions.
1189	The class '‹A.B›' cannot be imported because its leaf name is already resolved to the class that is being defined, '‹C.B›'. (For example, compiling class `util` generates error 1189 if the statement `import mx.util` appears in the util.as file.)
1190	The class '‹A.B›' cannot be imported because its leaf name is already resolved to a previously imported class '‹C.B›'. (For example, compiling `import jv.util` generates error 1190 if the statement `import mx.util` also appears in the AS file.)
1191	A class' instance variables may only be initialized to compile-time constant expressions.
1192	Class member functions cannot have the same name as a superclass' constructor function.
1193	The name of this class, '‹ClassName›', conflicts with the name of another class that was loaded.
1194	The superconstructor must be called first in the constructor body.
1195	The identifier '‹className›' will not resolve to built-in object '‹ClassName›' at runtime.
1196	The class '‹A.B.ClassName›' needs to be defined in a file whose relative path is ‹'A.B›'.
1197	The wildcard character '*' is misused in the ClassName '‹ClassName›'.

Error number	Message text
1198	The member function '<classname>' has a different case from the name of the class being defined, '<ClassName>', and will not be treated as the class constructor at runtime.
1199	The only type allowed for a for-in loop iterator is String.
1200	A setter function may not return a value.
1201	The only attributes allowed for constructor functions are public and private.

APPENDIX B
Deprecated Flash 4 operators

The following table lists Flash 4-only operators, which are deprecated in ActionScript 2.0. Do not use these operators unless you are publishing to Flash Player 4 and earlier.

Operator	Description	Associativity
`not`	Logical NOT	Right to left
`and`	Logical AND	Left to right
`or`	Logical OR (Flash 4)	Left to right
`add`	String concatenation (formerly &)	Left to right
`instanceof`	Instance of	Left to right
`lt`	Less than (string version)	Left to right
`le`	Less than or equal to (string version)	Left to right
`gt`	Greater than (string version)	Left to right
`ge`	Greater than or equal to (string version)	Left to right
`eq`	Equal (string version)	Left to right
`ne`	Not equal (string version)	Left to right

APPENDIX C
Keyboard Keys and Key Code Values

The following tables list all the keys on a standard keyboard and the corresponding ASCII key code values that are used to identify the keys in ActionScript:

Letters A to Z and standard numbers 0 to 9

The following table lists the keys on a standard keyboard for the letters A to Z and the numbers 0 to 9, with the corresponding ASCII key code values that are used to identify the keys in ActionScript:

Letter or number key	Key code
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77

Letter or number key	Key code
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

Keys on the numeric keypad

The following table lists the keys on a numeric keypad, with the corresponding ASCII key code values that are used to identify the keys in ActionScript:

Numeric keypad key	Key code
Numbpad 0	96
Numbpad 1	97
Numbpad 2	98
Numbpad 3	99
Numbpad 4	100

Numeric keypad key	Key code
Numbpad 5	101
Numbpad 6	102
Numbpad 7	103
Numbpad 8	104
Numbpad 9	105
Multiply	106
Add	107
Enter	13
Subtract	109
Decimal	110
Divide	111

Function keys

The following table lists the function keys on a standard keyboard, with the corresponding ASCII key code values that are used to identify the keys in ActionScript:

Function key	Key code
F1	112
F2	113
F3	114
F4	115
F5	116
F6	117
F7	118
F8	119
F9	120
F10	This key is reserved by the system and cannot be used in ActionScript.
F11	122
F12	123
F13	124
F14	125
F15	126

Other keys

The following table lists keys on a standard keyboard other than letters, numbers, numeric keypad keys, or function keys, with the corresponding ASCII key code values that are used to identify the keys in ActionScript:

Key	Key code
Backspace	8
Tab	9
Clear	12
Enter	13
Shift	16
Control	17
Alt	18
Caps Lock	20
Esc	27
Spacebar	32
Page Up	33
Page Down	34
End	35
Home	36
Left Arrow	37
Up Arrow	38
Right Arrow	39
Down Arrow	40
Insert	45
Delete	46
Help	47
Num Lock	144
; :	186
= +	187
- _	189
/ ?	191
` ~	192
[{	219
\ |	220

Key	Key code
]}	221
" '	222

APPENDIX D
Writing Scripts for Earlier Versions of Flash Player

ActionScript has changed considerably with the release of Macromedia Flash MX 2004 and Macromedia Flash MX Professional 2004. When you create content for Macromedia Flash Player 7, you can use the full power of ActionScript. You can still use Flash MX 2004 to create content for earlier versions of Flash Player, but you can't use every ActionScript element.

This chapter provides guidelines to help you write scripts that are syntactically correct for the player version you are targeting.

Note: You can review surveys for Flash Player version penetration on the Macromedia website; see www.macromedia.com/software/player_census/flashplayer/.

About targeting older versions of Flash Player

When you write scripts, use the Availability information for each element in *Flash ActionScript Language Reference* to determine if an element you want to use is supported by the Flash Player version you are targeting. You can also determine which elements you can use by showing the Actions toolbox; elements that are not supported for your target version appear in yellow.

If you create content for Flash Player 6 or 7, you should use ActionScript 2.0, which provides several important features that aren't available in ActionScript 1, such as improved compiler errors and more robust object-oriented programming capabilities.

For a review of differences in how certain features are implemented when publishing files for Flash Player 7 versus how the features are implemented in files published for earlier player versions, see "Porting existing scripts to Flash Player 7" on page 13.

To specify the player and ActionScript version you want to use when publishing a document, select File > Publish Settings, and then make your selections on the Flash tab. If you need to target Flash Player 4, see the next section.

Using Flash MX 2004 to create content for Flash Player 4

To use Flash MX 2004 to create content for Flash Player 4, specify Flash Player 4 on the Flash tab of the Publish Settings dialog box (File > Publish Settings).

Flash Player 4 ActionScript has only one basic primitive data type, which is used for numeric and string manipulation. When you write an application for Flash Player 4, you must use the deprecated string operators located in the Deprecated > Operators category in the Actions toolbox.

You can use the following Flash MX 2004 features when you publish for Flash Player 4:

- The array and object access operator (`[]`)
- The dot operator (`.`)
- Logical operators, assignment operators, and pre- and post-increment/decrement operators
- The modulo operator (`%`) and all methods and properties of the Math class

The following language elements are not supported natively by Flash Player 4. Flash MX 2004 exports them as series approximations, which creates results that are less numerically accurate. In addition, because of the inclusion of series approximations in the SWF file, these language elements need more room in Flash Player 4 SWF files than they do in Flash Player 5 or later SWF files.

- The `for`, `while`, `do..while`, `break`, and `continue` actions
- The `print()` and `printAsBitmap()` actions
- The `switch` action

For additional information, see "About targeting older versions of Flash Player" on page 319.

Using Flash MX 2004 to open Flash 4 files

Flash 4 ActionScript had only one true data type: string. It used different types of operators in expressions to indicate whether the value should be treated as a string or as a number. In subsequent releases of Flash, you can use one set of operators on all data types.

When you use Flash 5 or later to open a file that was created in Flash 4, Flash automatically converts ActionScript expressions to make them compatible with the new syntax. Flash makes the following data type and operator conversions:

- The = operator in Flash 4 was used for numeric equality. In Flash 5 and later, == is the equality operator and = is the assignment operator. Any = operators in Flash 4 files are automatically converted to ==.
- Flash automatically performs type conversions to ensure that operators behave as expected. Because of the introduction of multiple data types, the following operators have new meanings:

  +, ==, !=, <>, <, >, >=, <=

In Flash 4 ActionScript, these operators were always numeric operators. In Flash 5 and later, they behave differently, depending on the data types of the operands. To prevent any semantic differences in imported files, the `Number()` function is inserted around all operands to these operators. (Constant numbers are already obvious numbers, so they are not enclosed in `Number()`). For more information on these operators, see the operator table in "Operator precedence and associativity" on page 49 and "Deprecated Flash 4 operators" on page 311.

- In Flash 4, the escape sequence `\n` generated a carriage return character (ASCII 13). In Flash 5 and later, to comply with the ECMA-262 standard, `\n` generates a line-feed character (ASCII 10). An `\n` sequence in Flash 4 FLA files is automatically converted to `\r`.
- The `&` operator in Flash 4 was used for string addition. In Flash 5 and later, `&` is the bitwise AND operator. The string addition operator is now called `add`. Any `&` operators in Flash 4 files are automatically converted to `add` operators.
- Many functions in Flash 4 did not require closing parens; for example, `Get Timer`, `Set Variable`, `Stop`, and `Play`. To create consistent syntax, the `getTimer` function and all actions now require parentheses [()]. These parentheses are automatically added during the conversion.
- In Flash 5 and later, when the `getProperty` function is executed on a movie clip that doesn't exist, it returns the value `undefined`, not `0`. The statement `undefined == 0` is `false` in ActionScript after Flash 4 (in Flash 4, `undefined == 1`). In Flash 5 and later, solve this problem when converting Flash 4 files by introducing `Number()` functions in equality comparisons. In the following example, `Number()` forces `undefined` to be converted to `0` so the comparison will succeed:

```
getProperty("clip", _width) == 0
Number(getProperty("clip", _width)) == Number(0)
```

Note: If you used any Flash 5 or later keywords as variable names in your Flash 4 ActionScript, the syntax returns an error when you compile it in Flash MX 2004. To solve this problem, rename your variables in all locations. For information, see "Keywords and reserved words" on page 32 and "Naming a variable" on page 44.

Using slash syntax

Slash syntax (`/`) was used in Flash 3 and 4 to indicate the target path of a movie clip or variable. In slash syntax, slashes are used instead of dots and variables are preceded with a colon, as shown in the following example:

```
myMovieClip/childMovieClip:myVariable
```

To write the same target path in dot syntax (see "Dot syntax" on page 29), which is supported by Flash Player 5 and later, use the following syntax:

```
myMovieClip.childMovieClip.myVariable
```

Slash syntax was most commonly used with the `tellTarget` action, but its use is also no longer recommended. The `with` action is now preferred because it is more compatible with dot syntax. For more information, see `tellTarget` and `with` in *Flash ActionScript Language Reference.*

APPENDIX E
Object-Oriented Programming with ActionScript 1

The information in this appendix comes from the Macromedia Flash MX documentation and provides information on using the ActionScript 1 object model to write scripts. It is included here for the following reasons:

- If you want to write object-oriented scripts that support Flash Player 5, you must use ActionScript 1.
- If you already use ActionScript 1 to write object-oriented scripts and aren't ready to switch to ActionScript 2.0, you can use this appendix to find or review information you need while writing your scripts.

If you have never used ActionScript to write object-oriented scripts and don't need to target Flash Player 5, you should not use the information in this appendix because writing object-oriented scripts using ActionScript 1 is deprecated. Instead, for information on using ActionScript 2.0, see Chapter 10, "Creating Custom Classes with ActionScript 2.0," on page 247.

Note: Some examples in this appendix use the `Object.RegisterClass()` method. This method is supported only in Flash Player 6 and later; don't use this method if you are targeting Flash Player 5.

About ActionScript 1

ActionScript is an object-oriented programming language. Object-oriented programming uses *objects*, or data structures, to group together properties and methods that control the object's behavior or appearance. Objects let you organize and reuse code. After you define an object, you can refer to it by name without having to redefine it each time you use it.

A *class* is a generic category of objects. A class defines a series of objects that have common properties and can be controlled in the same ways. Properties are attributes that define an object, such as its size, position, color, transparency, and so on. Properties are defined for a class, and values for the properties are set for individual objects in the class. Methods are functions that can set or retrieve properties of an object. For example, you can define a method to calculate the size of an object. As with properties, methods are defined for an object class and then invoked for individual objects in the class.

ActionScript includes several built-in classes, including the MovieClip class, Sound class, and others. You can also create custom classes to define categories of objects for your applications.

Objects in ActionScript can be pure containers for data, or they can be graphically represented on the Stage as movie clips, buttons, or text fields. All movie clips are instances of the built-in MovieClip class, and all buttons are instances of the built-in Button class. Each movie clip instance contains all the properties (for example, `_height`, `_rotation`, `_totalframes`) and all the methods (for example, `gotoAndPlay()`, `loadMovie()`, `startDrag()`) of the MovieClip class.

To define a class, you create a special function called a *constructor function*. (Built-in classes have built-in constructor functions.) For example, if you want information about a bicycle rider in your application, you could create a constructor function, `Biker()`, with the properties `time` and `distance` and the method `getSpeed()`, which tells you how fast the biker is traveling:

```
function Biker(t, d) {
  this.time = t;
  this.distance = d;
  this.getSpeed = function() {return this.time / this.distance;};
}
```

In this example, you create a function that needs two pieces of information, or *parameters*, to do its job: `t` and `d`. When you call the function to create new instances of the object, you pass it the parameters. The following code creates instances of the object Biker called `emma` and `hamish`, and it traces the speed of the `emma` instance, using the `getSpeed()` method from the previous ActionScript:

```
emma = new Biker(30, 5);
hamish = new Biker(40, 5);
trace (emma.getSpeed()); //traces 6
```

In object-oriented scripting, classes can receive properties and methods from each other according to a specific order, which is called *inheritance*. You can use inheritance to extend or redefine the properties and methods of a class. A class that inherits from another class is called a *subclass*. A class that passes properties and methods to another class is called a *superclass*. A class can be both a subclass and a superclass.

An object is a complex data type containing zero or more properties and methods. Each property, like a variable, has a name and a value. Properties are attached to the object and contain values that can be changed and retrieved. These values can be of any data type: String, Number, Boolean, Object, MovieClip, or undefined. The following properties are of various data types:

```
customer.name = "Jane Doe";
customer.age = 30;
customer.member = true;
customer.account.currentRecord = 609;
customer.mcInstanceName._visible = true;
```

The property of an object can also be an object. In line 4 of the previous example, `account` is a property of the object `customer`, and `currentRecord` is a property of the object `account`. The data type of the `currentRecord` property is Number.

Creating a custom object in ActionScript 1

To create a custom object, you define a constructor function. A constructor function is always given the same name as the type of object it creates. You can use the keyword `this` inside the body of the constructor function to refer to the object that the constructor creates; when you call a constructor function, Flash passes `this` to the function as a hidden parameter. For example, the following code is a constructor function that creates a circle with the property `radius`:

```
function Circle(radius) {
  this.radius = radius;
}
```

After you define the constructor function, you must create an instance of the object. Use the `new` operator before the name of the constructor function, and assign a variable name to the new instance. For example, the following code uses the `new` operator to create a Circle object with a radius of 5 and assigns it to the variable `myCircle`:

```
myCircle = new Circle(5);
```

Note: An object has the same scope as the variable to which it is assigned.

Assigning methods to a custom object in ActionScript 1

You can define the methods of an object inside the object's constructor function. However, this technique is not recommended because it defines the method every time you use the constructor function. The following example creates the methods `getArea()` and `getDiameter()`: and traces the area and diameter of the constructed instance `myCircle` with a radius set to 55:

```
function Circle(radius) {
  this.radius = radius;
  this.getArea = function(){
    return Math.PI*this.radius*this.radius;
  };
  this.getDiameter = function() {
    return 2*this.radius;
  };
}
var myCircle = new Circle(55);
trace(myCircle.getArea());
trace(myCircle.getDiameter());
```

Each constructor function has a `prototype` property that is created automatically when you define the function. The `prototype` property indicates the default property values for objects created with that function. Each new instance of an object has a `__proto__` property that refers to the `prototype` property of the constructor function that created it. Therefore, if you assign methods to an object's `prototype` property, they are available to any newly created instance of that object. It's best to assign a method to the `prototype` property of the constructor function because it exists in one place and is referenced by new instances of the object (or class). You can use the `prototype` and `__proto__` properties to extend objects so that you can reuse code in an object-oriented manner. (For more information, see "Creating inheritance in ActionScript 1" on page 328.)

The following procedure shows how to assign an `getArea()` method to a custom Circle object.

To assign a method to a custom object:

1. Define the constructor function `Circle()`:

```
function Circle(radius) {
  this.radius = radius;
}
```

2. Define the `getArea()` method of the Circle object. The `getArea()` method calculates the area of the circle. In the following example, you can use a function literal to define the `getArea()` method and assign the `getArea` property to the circle's prototype object:

```
Circle.prototype.getArea = function () {
     return Math.PI * this.radius * this.radius;
};
```

3. The following example creates an instance of the Circle object:

```
var myCircle = new Circle(4);
```

4. Call the `getArea()` method of the new `myCircle` object using the following code:

```
var myCircleArea = myCircle.getArea();
trace(myCircleArea); //traces 50.265...
```

   ActionScript searches the `myCircle` object for the `getArea()` method. Because the object doesn't have an `getArea()` method, its prototype object `Circle.prototype` is searched for `getArea()`. ActionScript finds it, calls it, and traces `myCircleArea`.

Defining event handler methods in ActionScript 1

You can create an ActionScript class for movie clips and define the event handler methods in the prototype object of that new class. Defining the methods in the prototype object makes all the instances of this symbol respond the same way to these events.

You can also add an `onClipEvent()` or `on()` event handler action to an individual instance to provide unique instructions that run only when that instance's event occurs. The `onClipEvent()` and `on()` actions don't override the event handler method; both events cause their scripts to run. However, if you define the event handler methods in the prototype object and also define an event handler method for a specific instance, the instance definition overrides the prototype definition.

To define an event handler method in an object's prototype object:

1. Create a movie clip symbol and set the linkage ID to `theID` by selecting the symbol in the Library panel and selecting Linkage from the Library options menu.
2. In the Actions panel (Window > Development Panels > Actions), use the `function` action to define a new class, as shown in the following example:

```
// define a class
function myClipClass() {}
```

This new class is assigned to all instances of the movie clip that are added to the application by the Timeline or that are added to the application with the `attachMovie()` or `duplicateMovieClip()` method. If you want these movie clips to have access to the methods and properties of the built-in MovieClip object, you need to make the new class inherit from the MovieClip class.

3. Enter code, such as the following example:

```
// inherit from MovieClip class
myClipClass.prototype = new MovieClip();
```

   Now, the class `myClipClass` inherits all the properties and methods of the MovieClip class.

4. Enter code, such as the following example, to define the event handler methods for the new class:

```
// define event handler methods for myClipClass class
myClipClass.prototype.onLoad = function() {trace ("movie clip loaded");}
myClipClass.prototype.onEnterFrame = function() {trace ("movie clip entered
  frame");}
```

5. Select Window > Library to open the Library panel if it isn't already open.
6. Select the symbols that you want to associate with your new class, and select Linkage from the Library panel options menu.
7. In the Linkage Properties dialog box, select Export for ActionScript.
8. Enter an identifier in the Identifier text box.

   The identifier must be the same for all symbols that you want to associate with the new class. In the `myClipClass` example, the identifier is `theID`.

9. Enter code, such as the following example, in the Script pane:

```
// register class
Object.registerClass("theID", myClipClass);
this.attachMovie("theID","myName",1);
```

This step registers the symbol whose linkage identifier is `theID` with the class `myClipClass`. All instances of `myClipClass` have event handler methods that behave as defined in step 4. They also behave the same as all instances of the MovieClip class because you told the new class to inherit from the class MovieClip in step 3.

The complete code is shown in the following example:

```
function myClipClass(){}

myClipClass.prototype = new MovieClip();
myClipClass.prototype.onLoad = function(){
  trace("movie clip loaded");
}
myClipClass.prototype.onPress = function(){
  trace("pressed");
}

myClipClass.prototype.onEnterFrame = function(){
  trace("movie clip entered frame");
}
```

```
myClipClass.prototype.myfunction = function(){
  trace("myfunction called");
}

Object.registerClass("myclipID",myClipClass);
this.attachMovie("myclipID","clipName",3);
```

Creating inheritance in ActionScript 1

Inheritance is a means of organizing, extending, and reusing functionality. Subclasses inherit properties and methods from superclasses and add their own specialized properties and methods. For example, reflecting the real world, Bike would be a superclass and MountainBike and Tricycle would be subclasses of the superclass. Both subclasses contain, or *inherit*, the methods and properties of the superclass (for example, `wheels`). Each subclass also has its own properties and methods that extend the superclass (for example, the MountainBike subclass would have a `gears` property). You can use the elements `prototype` and `__proto__` to create inheritance in ActionScript.

All constructor functions have a `prototype` property that is created automatically when the function is defined. The `prototype` property indicates the default property values for objects created with that function. You can use the `prototype` property to assign properties and methods to a class. (For more information, see "Assigning methods to a custom object in ActionScript 1" on page 325.)

All instances of a class have a `__proto__` property that tells you the object from which they inherit. When you use a constructor function to create an object, the `__proto__` property is set to refer to the `prototype` property of its constructor function.

Inheritance proceeds according to a definite hierarchy. When you call an object's property or method, ActionScript looks at the object to see if such an element exists. If it doesn't exist, ActionScript looks at the object's `__proto__` property for the information (`myObject.__proto__`). If the property is not a property of the object's `__proto__` object, ActionScript looks at `myObject.__proto__.__proto__`, and so on.

The following example defines the constructor function `Bike()`:

```
function Bike(length, color) {
  this.length = length;
  this.color = color;
  this.pos = 0;
}
```

The following code adds the `roll()` method to the Bike class:

```
Bike.prototype.roll = function() {return this.pos += 20;};
```

Then, you can trace the position of the bike with the following code:

```
var myBike = new Bike(55, "blue");
trace(myBike.roll());   // traces 20.
trace(myBike.roll());   // traces 40.
```

Instead of adding `roll()` to the MountainBike class and the Tricycle class, you can create the MountainBike class with Bike as its superclass, as shown in the following example:

```
MountainBike.prototype = new Bike();
```

Now you can call the `roll()` method of MountainBike, as shown in the following example:

```
var myKona = new MountainBike(20, "teal");
trace(myKona.roll());   //traces 20
```

Movie clips do not inherit from each other. To create inheritance with movie clips, you can use `Object.registerClass()` to assign a class other than the MovieClip class to movie clips. See `Object.registerClass()` in *Flash ActionScript Language Reference*.

For more information on inheritance, see `Object.__proto__` and `super` in *Flash ActionScript Language Reference*. For information on `#initclip` and `#endinitclip` see "Applying new skins to a subcomponent" in *Using Components*.

Adding getter/setter properties to objects in ActionScript 1

You can create getter/setter properties for an object using the `Object.addProperty()` method.

A getter function is a function with no parameters. Its return value can be of any type. Its type can change between invocations. The return value is treated as the current value of the property.

A setter function is a function that takes one parameter, which is the new value of the property. For instance, if property x is assigned by the statement x = 1, the setter function is passed the parameter 1 of type Number. The return value of the setter function is ignored.

When Flash reads a getter/setter property, it invokes the getter function, and the function's return value becomes a value of `prop`. When Flash writes a getter/setter property, it invokes the setter function and passes it the new value as a parameter. If a property with the given name already exists, the new property overwrites it.

You can add getter/setter properties to prototype objects. If you add a getter/setter property to a prototype object, all object instances that inherit the prototype object inherit the getter/setter property. You can add a getter/setter property in one location, the prototype object, and have it propagate to all instances of a class (similar to adding methods to prototype objects). If a getter/setter function is invoked for a getter/setter property in an inherited prototype object, the reference passed to the getter/setter function is the originally referenced object, not the prototype object.

For more information, see "`Object.addProperty()`" in *Flash ActionScript Language Reference.*

The Debug > List Variables command in test mode supports getter/setter properties that you add to objects using `Object.addProperty()`. Properties that you add to an object in this way appear with other properties of the object in the Output panel. Getter/setter properties are identified in the Output panel with the prefix `[getter/setter]`. For more information on the List Variables command, see "Using the Output panel" on page 162.

Using Function object properties in ActionScript 1

You can specify the object to which a function is applied and the parameter values that are passed to the function, using the `call()` and `apply()` methods of the Function object. Every function in ActionScript is represented by a Function object, so all functions support `call()` and `apply()`. When you create a custom class using a constructor function, or when you define methods for a custom class using a function, you can invoke `call()` and `apply()` for the function.

Invoking a function using the Function.call() method in ActionScript 1

The `Function.call()` method invokes the function represented by a Function object.

In almost all cases, the function call operator (`()`) can be used instead of the `call()` method. The function call operator creates code that is concise and readable. The `call()` method is primarily useful when the `this` parameter of the function invocation needs to be explicitly controlled. Normally, if a function is invoked as a method of an object, within the body of the function, `this` is set to `myObject`, as shown in the following example:

```
myObject.myMethod(1, 2, 3);
```

In some situations, you might want `this` to point somewhere else; for instance, if a function must be invoked as a method of an object but is not actually stored as a method of that object, as shown in the following example:

```
myObject.myMethod.call(myOtherObject, 1, 2, 3);
```

You can pass the value `null` for the *`thisObject`* parameter to invoke a function as a regular function and not as a method of an object. For example, the following function invocations are equivalent:

```
Math.sin(Math.PI / 4)
Math.sin.call(null, Math.PI / 4)
```

For more information, see "`Function.call()`" in *Flash ActionScript Language Reference.*

To invoke a function using the Function.call method:

- Use the following syntax:

  `myFunction``.call(`*`thisObject, parameter1, ..., parameterN`*`)`

  The method takes the following parameters:

  - The parameter *`thisObject`* specifies the value of `this` within the function body.
  - The parameters *`parameter1...`*, *`parameterN`* specify parameters to be passed to *`myFunction`*. You can specify zero or more parameters.

Specifying the object to which a function is applied using Function.apply() in ActionScript 1

The `Function.apply()` method specifies the value of `this` to be used within any function that ActionScript calls. This method also specifies the parameters to be passed to any called function.

The parameters are specified as an Array object. This is often useful when the number of parameters to be passed is not known until the script actually executes.

For more information, see `Function.apply()` in *Flash ActionScript Language Reference.*

To specify the object to which a function is applied using Function.apply():

- Use the following syntax:

  myFunction`.apply(`*thisObject*`, `*argumentsObject*`)`

  The method takes the following parameters:

  - The parameter *thisObject* specifies the object to which *myFunction* is applied.
  - The parameter *argumentsObject* defines an array whose elements are passed to *myFunction* as parameters.

INDEX

C

D

E

F

K

L

M

N

O

T

U

V

W

X