macromedia®

FLASH™ 5

ActionScript Reference Guide

macromedia™

Trademarks

Afterburner, AppletAce, Attain, Attain Enterprise Learning System, Attain Essentials, Attain Objects for Dreamweaver, Authorware, Authorware Attain, Authorware Interactive Studio, Authorware Star, Authorware Synergy, Backstage, Backstage Designer, Backstage Desktop Studio, Backstage Enterprise Studio, Backstage Internet Studio, Design in Motion, Director, Director Multimedia Studio, Doc Around the Clock, Dreamweaver, Dreamweaver Attain, Drumbeat, Drumbeat 2000, Extreme 3D, Fireworks, Flash, Fontographer, FreeHand, FreeHand Graphics Studio, Generator, Generator Developer's Studio, Generator Dynamic Graphics Server, Knowledge Objects, Knowledge Stream, Knowledge Track, Lingo, Live Effects, Macromedia, Macromedia M Logo & Design, Macromedia Flash, Macromedia Xres, Macromind, Macromind Action, MAGIC, Mediamaker, Object Authoring, Power Applets, Priority Access, Roundtrip HTML, Scriptlets, SoundEdit, ShockRave, Shockmachine, Shockwave, Shockwave Remote, Shockwave Internet Studio, Showcase, Tools to Power Your Ideas, Universal Media, Virtuoso, Web Design 101, Whirlwind and Xtra are trademarks of Macromedia, Inc. and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words or phrases mentioned within this publication may be trademarks, servicemarks, or tradenames of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

This guide contains links to third-party Web sites that are not under the control of Macromedia, and Macromedia is not responsible for the content on any linked site. If you access a third-party Web site mentioned in this guide, then you do so at your own risk. Macromedia provides these links only as a convenience, and the inclusion of the link does not imply that Macromedia endorses or accepts any responsibility for the content on those third-party sites.

Apple Disclaimer

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Part Number ZFL50M200

Acknowledgments

Project Management: Erick Vera

Writing: Jody Bleyle, Mary Burger, Louis Dobrozensky, Stephanie Gowin, Marcelle Taylor, and Judy Walthers Von Alten

Editing: Peter Fenczik, Rosana Francescato, Ann Szabla

Multimedia: George Brown, John "Zippy" Lehnus, and Noah Zilberberg

Print and Help Design: Chris Basmajian and Noah Zilberberg

Production: Chris Basmajian and Rebecca Godbois

Special thanks: Jeremy Clark, Brian Dister and the entire Flash Development team, Margaret Dumas, Kipling Inscore, Alyn Kelley and the entire Flash QA team, Pete Santangeli, Cyn Taylor, and Eric Wittman

First Edition: July 2000

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

CONTENTS

INTRODUCTION
Getting Started 7

What's new in Flash 5 ActionScript 7
Using Flash Help for actions 10

CHAPTER 1
Understanding ActionScript 11

About scripting in ActionScript 12
ActionScript terminology 19
Deconstructing a sample script 22
Using the Actions panel 24
Assigning actions to objects 33
Assigning actions to frames 35

CHAPTER 2
Writing Scripts with ActionScript 37

Using ActionScript's syntax 37
About data types 42
About variables 46
Using operators to manipulate values in expressions 51
Using actions 58
Controlling flow in scripts 60
Using predefined functions 63
Creating custom functions 65
Using predefined objects 68
Using custom objects 72
Opening Flash 4 files 75
Using Flash 5 to create Flash 4 content 77

CHAPTER 3
Creating Interaction with ActionScript 79

Creating a custom cursor 80
Getting the mouse position 82
Capturing keypresses 84
Creating a scrolling text field 86
Setting color values 88
Creating sound controls 90
Detecting collisions 94

CHAPTER 4
Working with Movie Clips 97

About multiple Timelines 98
Using actions and methods to control Timelines 111
Creating smart clips 119

CHAPTER 5
Integrating Flash with Web Applications 125

Sending and loading variables to and from a remote file 126
Creating forms 137
Sending messages to and from the Flash Player 141

CHAPTER 6
Troubleshooting ActionScript 145

Authoring and troubleshooting guidelines 145
Using the Debugger 148
Using the Output window 154

CHAPTER 7
ActionScript Dictionary 157

Sample entry for most ActionScript elements 158
Sample entry for objects 159
Contents of the dictionary 160

APPENDIX A
Operator Precedence and Associativity. 431

APPENDIX B
Keyboard Keys and Key Code Values 435

Letters A to Z and standard numbers 0 to 9 .435
Keys on the numeric keypad 437
Function keys 438
Other keys 439

APPENDIX C
Error Messages 441

INDEX 445

INTRODUCTION
Getting Started

ActionScript is Flash's scripting language. You can use ActionScript to control objects in Flash movies to create navigation and interactive elements and to extend Flash to create highly interactive movies and Web applications.

What's new in Flash 5 ActionScript

Flash 5 ActionScript offers exciting new features for creating immersive, interactive Web sites with sophisticated games, forms, surveys, and real-time interactivity such as chat systems.

Flash 5 ActionScript has many new features and syntax conventions that make it similar to the core JavaScript programming language. This manual will explain basic programming concepts such as functions, variables, statements, operators, conditionals, and loops. Chapter 7 of this manual, "ActionScript Dictionary," contains a detailed entry for every ActionScript element.

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

The European Computers Manufacturers Association (ECMA) wrote a document called ECMA-262 that was derived from JavaScript to serve as the international standard for the JavaScript language. ActionScript is based on the ECMA-262 specification, which is available from http://www.ecma.ch.

Netscape DevEdge Online has a JavaScript Developer Central (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*, located at http://developer.netscape.com/docs/manuals/js/core/jsguide/index.html.

Differences between ActionScript and JavaScript

You don't need to know JavaScript to use and learn ActionScript. If you know JavaScript, however, ActionScript will appear familiar to you. Some of the differences between ActionScript and JavaScript are as follows:

- ActionScript does not support browser-specific objects such as Document, Window, and Anchor.
- ActionScript does not completely support all of the JavaScript predefined objects.
- ActionScript supports syntax constructs that are not permitted in JavaScript (for example, the `tellTarget` and `ifFrameLoaded` actions and slash syntax).
- ActionScript does not support some JavaScript syntax constructs, such as `switch`, `continue`, `try`, `catch`, `throw`, and `statement` labels.
- ActionScript does not support the JavaScript `Function` constructor.
- In ActionScript, the `eval` action can only perform variable references.
- In JavaScript, `toString` of `undefined` is `undefined`. In Flash 5, for Flash 4 compatibility, `toString` of `undefined` is `" "`.
- In JavaScript, evaluating `undefined` in a numeric context results in `NaN`. In Flash 5, for Flash 4 compatibility, evaluating `undefined` results in `0`.
- ActionScript does not support Unicode; it supports ISO-8859-1 and Shift-JIS character sets.

Text editing

You can enter scripts directly into the Actions panel in Expert Mode. You can also choose elements from a pop-up menu or a Toolbox list just like you did in Flash 4.

Dot syntax

You can use dot syntax to get and set the properties and methods of an object, including movie clip instances and variables. You can use dot syntax instead of the slash syntax used in Flash 4. Slash syntax is no longer preferred, but it is still supported by the Flash Player.

Data types

Flash 5 ActionScript supports the following data types: string, number, boolean, object, and movie clip. Multiple data types allow you to use different types of information in ActionScript. For example, you can create arrays and associative arrays.

Local variables

You can declare local variables that expire at the end of the action list or function call. This allows you to manage memory and reuse variable names. Flash 4 variables were all permanent—even temporary variables like loop counters remained in the movie until the movie ended.

User-defined functions

You can define functions with parameters that return values. This allows you to reuse blocks of code in your scripts. In Flash 4, you could reuse code by using the `call` action, but you couldn't pass parameters or return values.

Predefined objects

You can use predefined objects to access and manipulate certain types of information. The following are a few of the predefined objects:

- The Math object features a full complement of built-in mathematical constants and functions, such as `E` (Euler's constant), `cos` (Cosine), and `atan` (Arctangent).
- The Date object allows you to get information about the date and time on whatever system is running the Flash Player.
- The Sound object allows you to add sounds to a movie and control sounds in a movie as it plays. For example, you can adjust volume (`setVolume`), or balance (`setPan`).
- The Mouse object allows you to hide the standard cursor so that you can use a custom cursor.
- The MovieClip object allows you to control movie clips without using a wrapper action such as `tellTarget`. You can call a method such as `play`, `loadMovie`, or `duplicateMovieClip` from an instance name by using dot syntax (for example, `myMovieClip.play()`).

Clip actions

You can use the `onClipEvent` action to assign actions directly to movie clip instances on the Stage. The `onClipEvent` action has events such as `load`, `enterFrame`, `mouseMove`, and `data` that allow you to create new kinds of advanced interactivity.

New actions

You can use new actions such as `do..while` and `for` to create complex loops. Other new actions are implemented as methods of the MovieClip object; for example, `getBounds`, `attachMovie`, `hitTest`, `swapDepths`, and `globalToLocal`.

Smart Clips

Smart Clips have internal scripts that you, or another developer, can change without using the Actions panel. You can pass values to a Smart Clip through clip parameters that you can define in the Library.

Debugger

The Debugger allows you to view and change variable and property values in a movie playing in Test-movie mode, the stand-alone Flash Player, or a Web browser. This allows you to easily find problems in your ActionScript.

XML support

The predefined XML object allows you to convert ActionScript to XML documents and pass them to server-side applications. You can also use the XML object to load XML documents into a Flash movie and interpret them. The predefined XMLSocket object allows you to create a continuous server connection to pass XML data for real-time applications.

Using Flash Help for actions

Flash 5 contains context-sensitive help for each action available in the Actions panel. While you're creating scripts, you can get information on the actions you're using.

To get help on actions:

1. In the Actions panel, select an action in the Toolbox list.
2. Click the Help button at the top of the panel.

   The topic related to the action appears in the browser.

CHAPTER 1
Understanding ActionScript

ActionScript, Flash's scripting language, adds interactivity to a movie. You can set up your movie so that user events, such as button clicks and keypresses, trigger scripts that tell the movie what action to perform. For example, you can write a script that tells Flash to load different movies into the Flash Player depending on which navigation button a user chooses.

Think of ActionScript as a tool that allows you to create a movie that behaves exactly as you want. You don't need to understand every possible use of the tool to begin scripting; if you have a clear goal, you can start building scripts with simple actions. You can incorporate new elements of the language as you learn them to accomplish more complicated tasks.

This chapter introduces you to ActionScript as an object-oriented scripting language and provides an overview of ActionScript terms. It also deconstructs a sample script so that you can begin to focus on the bigger picture.

This chapter also introduces you to the Actions panel, where you can build scripts by selecting ActionScript elements or entering text into the Script window.

About scripting in ActionScript

You can start writing simple scripts without knowing much about ActionScript. All you need is a goal; then it's just a matter of picking the right actions. The best way to learn how simple ActionScript can be is to create a script. The following steps attach a script to a button that changes the visibility of a movie clip.

To change the visibility of a movie clip:

1. Choose Window > Common Libraries > Buttons, and then choose Window > Common Libraries > Movie Clips. Place a button and a movie clip on the Stage.
2. Select the movie clip instance on the Stage, and choose Window > Panels > Instance Properties.
3. In the Name field, enter **testMC**.
4. Select the button on the Stage, and choose Window > Actions to open the Actions panel.
5. In the Object Actions panel, click the Actions category to open it.
6. Double-click the `setProperty` action to add it to the Actions list.
7. From the Property pop-up menu, choose `_visible` (Visibility).
8. For the Target parameter, enter **testMC**.
9. For the Value parameter, enter **0**.

   The code should look like this:

   ```
   on (release) {
       setProperty ("testMC", _visible, false);
   }
   ```
10. Choose Control > Test Movie and click the button to see the movie clip disappear.

ActionScript is an object-oriented scripting language. This means that actions control objects when a particular event occurs. In this script, the event is the release of the mouse, the object is the movie clip instance `testMC`, and the action is `setProperty`. When the user clicks the onscreen button, a `release` event triggers a script that sets the `_visible` property of the object `MC` to `false` and causes the object to become invisible.

You can use the Actions panel to guide you through setting up simple scripts. To use the full power of ActionScript, it is important to understand how the language works: the concepts, elements, and rules that the language uses to organize information and create interactive movies.

This section explains the ActionScript workflow, the fundamental concepts of object-oriented scripting, Flash objects, and script flow. It also describes where scripts reside in a Flash movie.

About planning and debugging scripts

When you write scripts for an entire movie, the quantity and variety of scripts can be large. Deciding which actions to use, how to structure scripts effectively, and where scripts should be placed requires careful planning and testing, especially as the complexity of your movie grows.

Before you begin writing scripts, formulate your goal and understand what you want to achieve. This is as important—and typically as time consuming—as developing storyboards for your work. Start by writing out what you want to happen in the movie, as in this example:

- I want to create my whole site using Flash.
- Site visitors will be asked for their name, which will be reused in messages throughout the site.
- The site will have a draggable navigation bar with buttons that link to each section of the site.
- When a button is clicked, the new section will fade in to the center of the Stage.
- One scene will have a contact form with the user's name already filled in.

When you know what you want, you can build the objects you need and write the scripts to control those objects.

Getting scripts to work the way you want takes time—often more than one cycle of writing, testing, and debugging. The best approach is to start simple and test your work frequently. When you get one part of a script working, choose Save As to save a version of the file (for example, myMovie01.fla) and start writing the next part. This approach will help you identify bugs efficiently and ensure that your ActionScript is solid as you write more complex scripts.

About object-oriented scripting

In object-oriented scripting, you organize information by arranging it into groups called *classes*. You can create multiple instances of a class, called *objects,* to use in your scripts. You can use ActionScript's predefined classes and create your own.

When you create a class, you define all the *properties* (characteristics) and *methods* (behaviors) of each object it creates, just as real-world objects are defined. For example, a person has properties such as gender, height, and hair color and methods such as talk, walk, and throw. In this example, "person" is a class and each individual person is an object, or an *instance* of that class.

Objects in ActionScript can contain data or they can be graphically represented on the Stage as movie clips. All movie clips are instances of the predefined class MovieClip. 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*; predefined classes have constructor functions that are already defined. For example, if you want information about a bicycle rider in your movie, you could create a constructor function, `Biker`, with the properties `time` and `distance` and the method `rate`, which tells you how fast the biker is traveling:

```
function Biker(t, d) {
   this.time = t;
   this.distance = d;
}
function Speed() {
   return this.time / this.distance;
}
Biker.prototype.rate = Speed;
```

You could then create copies—that is, instances—of the class. The following code creates instances of the object `Biker` called `emma` and `hamish`.

```
emma = new Biker(30, 5);
hamish = new Biker(40, 5);
```

Instances can also communicate with each other. For the `Biker` object, you could create a method called `shove` that lets one biker shove another biker. (The instance `emma` could call its `shove` method if `hamish` got too close.) To pass information to a method, you use parameters (arguments): for example, the `shove` method could take the parameters *`who`* and *`howFar`*. In this example `emma` shoves `hamish` 10 pixels:

```
emma.shove(hamish, 10);
```

In object-oriented scripting, classes can receive properties and methods from each other according to a specific order; this 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.

About the MovieClip object

ActionScript's predefined classes are called *objects*. Each object allows you to access a certain type of information. For example, the Date object has methods (for example, `getFullYear`, `getMonth`), that allow you to read information from the system clock. The Sound object has methods (for example, `setVolume`, `setPan`) that allow you to control a sound in a movie. The MovieClip object has methods that allow you to control movie clip instances (for example, `play`, `stop`, and `getURL`) and get and set information about their properties (for example, `_alpha`, `_framesloaded`, `_visible`).

Movie clips are the most important objects of a Flash movie because they have Timelines that run independently of each other. For example, if the main Timeline only has one frame and a movie clip in that frame has ten frames, each frame in the movie clip will still play. This allows instances to act as autonomous objects that can communicate with each other.

Movie clip instances each have a unique instance name so that you can target them with an action. For example, you may have multiple instances on the Stage (for example, `leftClip` and `rightClip`) and only want one to play at a time. To assign an action that tells one particular instance to play, you need to use its name. In the following example, the movie clip's name is `leftClip`:

```
leftClip.play();
```

Instance names also allow you to duplicate, remove, and drag movie clips while a movie plays. The following example duplicates the instance `cartItem` to fill out a shopping cart with the number of items purchased:

```
onClipEvent(load) {
   do {
      duplicateMovieClip("cartItem", "cartItem" + i, i);
      i  = i + 1;
   } while (i <= numberItemsPur);
}
```

Movie clips have properties whose values you can set and retrieve dynamically with ActionScript. Changing and reading these properties can change the appearance and identity of a movie clip and is the key to creating interactivity. For example, the following script uses the `setProperty` action to set the transparency (alpha setting) of the `navigationBar` instance to 10:

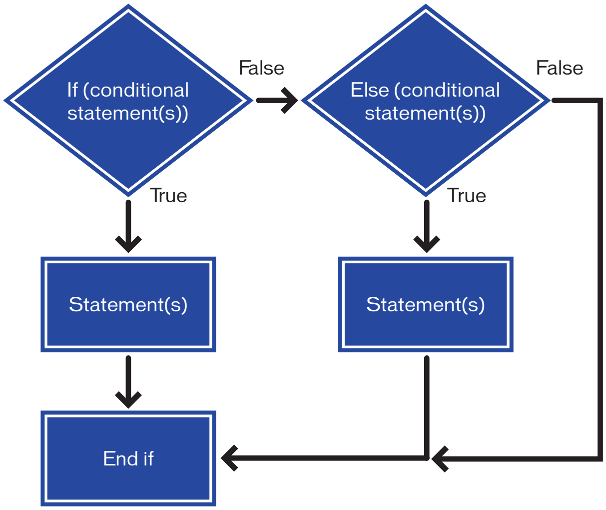
```
setProperty("navigationBar", _alpha, 10);
```

For more information about other types of objects, see "Using predefined objects" on page 68.

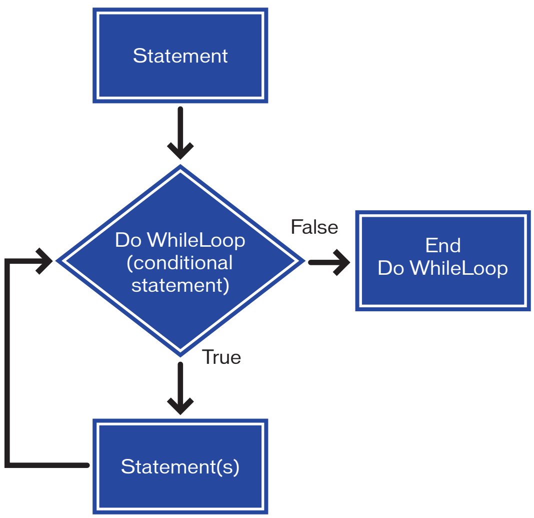
How scripts flow

ActionScript follows a logical flow. Flash executes ActionScript statements starting with the first statement and continuing in order until it reaches the final statement or a statement that instructs ActionScript to go somewhere else.

Some actions that send ActionScript somewhere other than the next statement are `if` statements, `do...while` loops, and the `return` action.



A flow chart of the if..else action



A flow chart of the do..while action

An `if` statement is called a conditional statement or a "logical branch" because it controls the flow of a script based on the evaluation of a certain condition. For example, the following code checks to see if the value of the `number` variable is less than or equal to 10. If the check returns `true` (for example, the value of `number` is 5), the variable `alert` is set and displays its value in an input text field, as in the following:

```
if (number <= 10) {
    alert = "The number is less than or equal to 10";
}
```

You can also add `else` statements to create a more complicated conditional statement. In the following example, if the condition returns `true` (for example, the value of `number` is 3), the statement between the first set of curly braces runs and the `alert` variable is set in the second line. If the condition returns `false` (for example, the value of `number` is 30), the first block of code is skipped and the statement between the curly braces after the `else` statement runs, as in the following:

```
if (number <= 10) {
    alert = "The number is less than or equal to 10";
} else {
    alert = "The number is greater than 10";
}
```

For more information, see "Using "if" statements" on page 60.

Loops repeat an action a certain number of times or until a certain condition is met. In the following example, a movie clip is duplicated five times:

```
i = 0;
do {
    duplicateMovieClip ("myMovieClip", "newMovieClip" + i, i);
    newName = eval("newMovieClip" + i);
   setProperty(newName, _x, getProperty("myMovieClip", _x) + (i *
5));
    i = i + 1;
} while (i <= 5);
```

For detailed information, see "Repeating an action" on page 61.

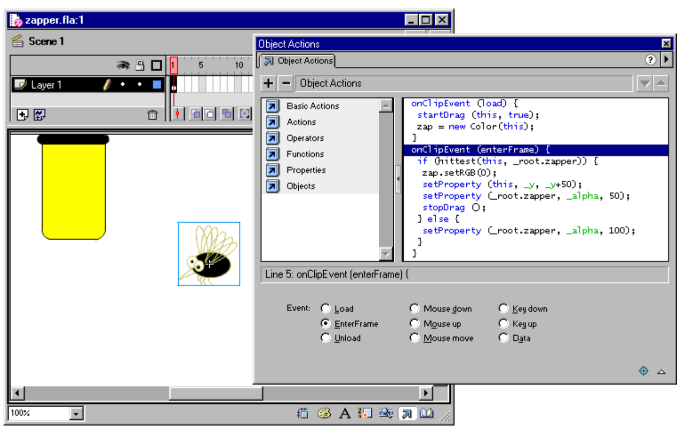
Controlling when ActionScript runs

When you write a script, you use the Actions panel. The Actions panel allows you to attach the script to a frame on the main Timeline or the Timeline of any movie clip, or to either a button or movie clip on the Stage.

Flash executes actions at different times, depending on what they're attached to:

- Actions attached to a frame are executed when the playhead enters that frame.
- Actions attached to a button are enclosed in an `on` handler action.
- Actions attached to a movie clip are enclosed in an `onClipEvent` handler action.

The `onClipEvent` and `on` actions are called handlers because they "handle" or manage an event. (An event is an occurrence such as a mouse movement, a keypress, or a movie clip being loaded.) Movie clip and button actions execute when the event specified by the handler occurs. You can attach more than one handler to an object if you want actions to execute when different events happen. For more information, see Chapter 3, "Creating Interactivity with ActionScript".



Several `onClipEvent` handlers attached to a movie clip on the Stage

ActionScript terminology

Like any scripting language, ActionScript uses specific terminology according to specific rules of syntax. The following list provides an introduction to important ActionScript terms in alphabetical order. These terms and the syntax that governs them are discussed in more detail in Chapter 2, "Writing Scripts with ActionScript."

Actions are statements that instruct a movie to do something while it is playing. For example, `gotoAndStop` sends the playhead to a specific frame or label. In this book, the terms *action* and *statement* are interchangeable.

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

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

Classes are data types that you can create to define a new type of object. To define a class of object, you create a constructor function.

Constants are elements that don't change. For example, the constant `TAB` always has the same meaning. Constants are useful for comparing values.

Constructors are functions that you use to define the properties and methods of a class. For example, the following code creates a new Circle class by creating a constructor function called `Circle`:

```
function Circle(x, y, radius){
   this.x = x;
   this.y = y;
   this.radius = radius;
}
```

Data types are a set of values and the operations that can be performed on them. String, number, `true` and `false` (Boolean) values, object, and movie clip are the ActionScript data types. For more details on these language elements, see "About data types" on page 42.

Events are actions that occur while a movie 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 at the keyboard.

Expressions are any parts of a statement that produce a value. For example, `2 + 2` is an expression.

Functions are blocks of reusable code that can be passed arguments (parameters) and can return a value. For example, the `getProperty` function is passed the name of a property and the instance name of a movie clip, and it returns the value of the property. The `getVersion` function returns the version of the Flash Player currently playing the movie.

Handlers are special actions that "handle" or manage an event such as `mouseDown` or `load`. For example, `on (onMouseEvent)` and `onClipEvent` are ActionScript handlers.

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 belong to a certain class. Each instance of a class contains all the properties and methods of that class. All movie clips are instances with properties (for example, `_alpha`, and `_visible`) and methods (for example, `gotoAndPlay`, and `getURL`) of the MovieClip class.

Instance names are unique names that allow you to target movie clip instances in scripts. For example, a master symbol in the Library could be called `counter` and the two instances of that symbol in the movie could have the instance names `scorePlayer1` and `scorePlayer2`. The following code sets a variable called `score` inside each movie clip instance by using instance names:

```
_root.scorePlayer1.score += 1
_root.scorePlayer2.score -= 1
```

Keywords are reserved words that have special meaning. For example, `var` is a keyword used to declare local variables.

Methods are functions assigned to an object. After a function is assigned, it can be called as a method of that object. For example, in the following code, `clear` becomes a method of the `controller` object:

```
function Reset(){
   x_pos = 0;
   x_pos = 0;
}
controller.clear = Reset;
controller.clear();
```

Objects are collections of properties; each object has its own name and value. Objects allow you to access a certain type of information. For example, the predefined Date object 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.

Target paths are hierarchical addresses of movie clip instance names, variables, and objects in a movie. You can name a movie clip instance in the Instance panel. 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. For example, the following statement is the target path to the variable `volume` inside the movie clip `stereoControl`:

```
_root.stereoControl.volume
```

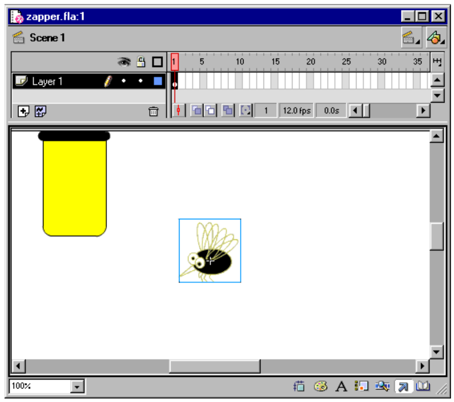
Properties are attributes that define an object. For example, `_visible` is a property of all movie clips that defines whether the movie clip is visible or hidden.

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:

```
x = 5;
name = "Lolo";
customer.address = "66 7th Street";
c = new Color(mcinstanceName);
```

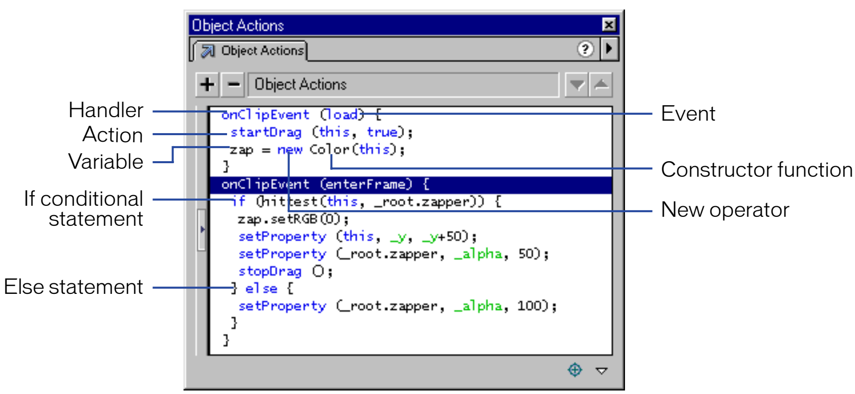
Deconstructing a sample script

In this sample movie, when a user drags the bug to the bug zapper, the bug turns black and falls and the bug zapper flashes. The movie is one frame long and contains two objects, the bug movie clip instance and the zapper movie clip instance. Each movie clip also contains one frame.



The bug and zapper movie clip instances on the Stage in frame 1

There is only one script in the movie; it's attached to the `bug` instance, as in the Object Actions panel below:



The Object Actions panel with the script attached to the `bug` *instance*

Both objects have to be movie clips so you can give them instance names in the Instance panel and manipulate them with ActionScript. The bug's instance name is `bug` and the zapper's instance name is `zapper`. 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 calls it.

There are two `onClipEvent` handlers with two different events: `load` and `enterFrame`. The actions in the `onClipEvent(load)` statement only execute once, when the movie loads. The actions in the `onClipEvent(enterFrame)` statement execute every time the playhead enters a frame. Even in a one-frame movie, the playhead still enters that frame repeatedly and the script executes repeatedly. The following actions occur within each `onClipEvent` handler:

onClipEvent(load) A `startDrag` action makes the bug movie clip draggable. An instance of the Color object is created with the `new` operator and the Color constructor function, `Color`, and assigned to the variable `zap`:

```
onClipEvent (load) {
   startDrag (this, true);
   zap = new Color(this);
}
```

onClipEvent(enterFrame) A conditional `if` statement evaluates a `hitTest` action to check whether the bug instance (`this`) is touching the bug zapper instance (`_root.zapper`). There are two possible outcomes of the evaluation, `true` or `false`:

```
onClipEvent (enterFrame) {
   if (this.hitTest(_root.zapper)) {
      zap.setRGB(0);
      setProperty (_target, _y, _y+50);
      setProperty (_root.zapper, _alpha, 50);
      stopDrag ();
   } else {
      setProperty (_root.zapper, _alpha, 100);
   }
}
```

If the `hitTest` action returns `true`, the `zap` object created by the `load` event is used to set the bug's color to black. The bug's *y* property (`_y`) is set to itself plus `50` so that the bug falls. The zapper's transparency (`_alpha`) is set to `50` so that it dims. The `stopDrag` action stops the bug from being draggable.

If the `hitTest` action returns `false`, the action following the `else` statement runs and the bug zapper's `_alpha` value is set to `100`. This makes the bug zapper appear to flash as its `_alpha` value goes from an initial state (100) to a zapped state (50) and back to an initial state. The `hitTest` action returns `false` and the `else` statements execute after the bug has been zapped and fallen.

To see the movie play, see *Flash Help*.

Using the Actions panel

The Actions panel lets you create and edit actions for an object or frame using two different editing modes. You can select prewritten actions from the Toolbox list, drag and drop actions, and use buttons to delete or rearrange actions. In Normal Mode you can write actions using parameter (argument) fields that prompt you for the correct arguments. In Expert Mode you can write and edit actions directly in a text box, much like writing script with a text editor.

To display the Actions panel:

Choose Window > Actions.

Selecting an instance of a button or movie clip makes the Actions panel active. The Actions panel title changes to Object Actions if a button or movie clip is selected, and to the Frame Actions panel if a frame is selected.

To select an editing mode:

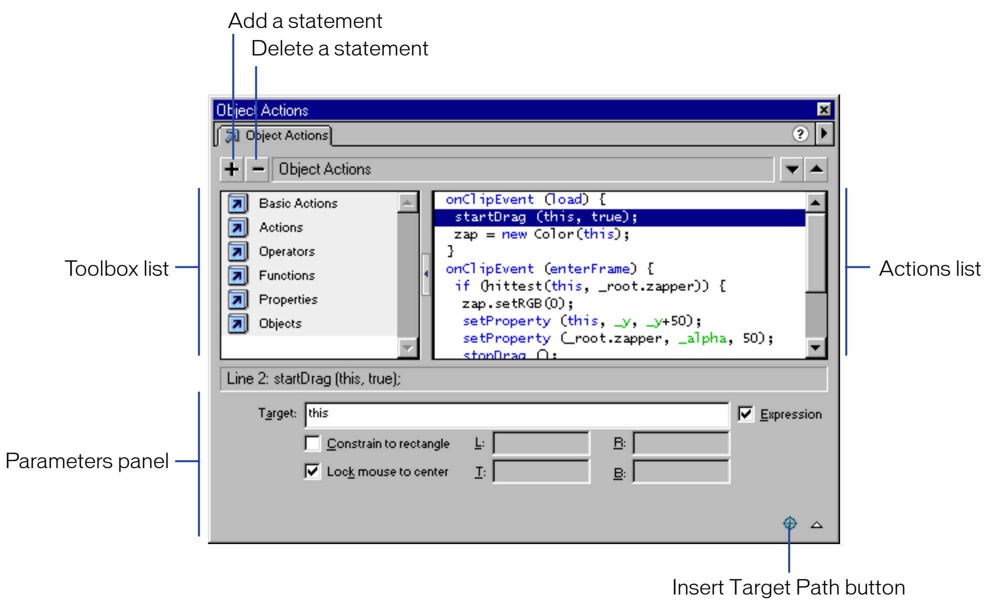
1. With the Actions panel displayed, click the arrow in the upper right corner of the panel to display the pop-up menu.
2. Choose Normal Mode or Expert Mode from the pop-up menu.

   Each script maintains its own mode. For example, you can script one instance of a button in Normal Mode, and another in Expert Mode. Switching between the selected button then switches the panel's mode state.

Normal Mode

In Normal Mode you create actions by selecting actions from a list on the left side of the panel, called the Toolbox list. The Toolbox list contains Basic Actions, Actions, Operators, Functions, Properties, and Objects categories. The Basic Actions category contains the simplest Flash actions and is only available in Normal Mode. The selected actions are listed on the right side of the panel, in the Actions list. You can add, delete, or change the order of action statements; you can also enter parameters (arguments) for actions in parameter fields at the bottom of the panel.

In Normal Mode you can use the controls in the Actions panel to delete or change the order of statements in the Actions list. These controls are especially useful for managing frame or button actions that have several statements.



The Actions panel in Normal Mode.

To select an action:

1 Click an Actions category in the toolbox to display the actions in that category.

2 Double-click an action or drag it to the Script window.

To use the Parameters fields:

1 Click the Parameters button in the lower right corner of the Actions panel to display the fields.

2 Select the action and enter new values in the Parameters fields to change parameters of existing actions.

To insert a movie clip target path:

1 Click the Target Path button in the lower right corner of the Actions panel to display the Insert Target Path dialog box.

2 Select a movie clip from the display list.

To move a statement up or down the list:

1 Select a statement in the Actions list.

2 Click the Up or Down Arrow buttons.

To delete an action:

1 Select a statement in the Actions list.

2 Click the Delete (-) button.

To change the parameters of existing actions:

1 Select a statement in the Actions list.

2 Enter new values in the Parameters fields.

To resize the Toolbox or Actions list, do one of the following:

- Drag the vertical splitter bar that appears between the Toolbox and Actions list.
- Double-click the splitter bar to collapse the Toolbox list; double-click the bar again to redisplay the list.
- Click the Left or Right Arrow button on the splitter bar to expand or collapse the list.

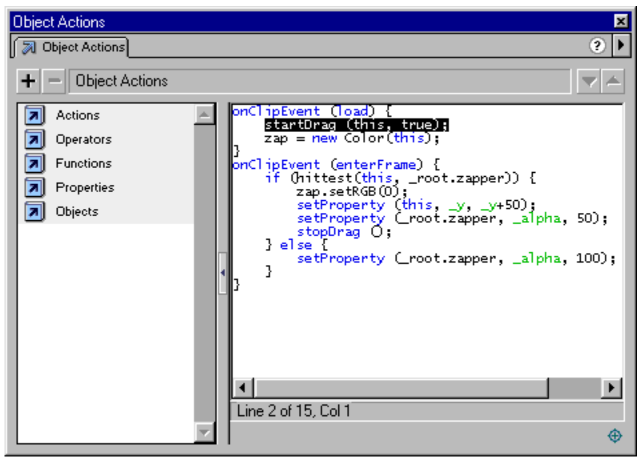
When the Toolbox list is hidden, you can still access its items using the Add (+) button in the upper left of the Actions panel.

Expert Mode

In Expert Mode you create actions by entering ActionScript into the text box on the right side of the panel or by selecting actions from the Toolbox list on the left. You edit actions, enter parameters for actions, or delete actions directly in the text box, much like creating script in a text editor.

Expert Mode lets advanced ActionScript users edit their scripts with a text editor, as they would JavaScript or VBScript. Expert Mode differs from Normal Mode in these ways:

- Selecting an item using the Add (+) button or Toolbox list inserts the item in the text-editing area.
- No parameter fields appear.
- In the button panel, only the Add (+) button works.
- The Up and Down Arrow buttons remain inactive.



The Actions panel in Expert Mode

Switching between editing modes

Changing editing modes while writing a script can change the formatting of the script. For that reason, it is best to use one editing mode per script.

When you switch from Normal to Expert Mode, indentation and formatting is maintained. Although you can convert Normal Mode scripts with errors to Expert Mode, you cannot export the scripts until the errors are fixed.

Switching from Expert to Normal Mode is slightly more complex:

- When you switch to Normal Mode, Flash reformats the script and strips any white space and indentation you've added.
- If you switch to Normal Mode and then back to Expert Mode, Flash reformats the script according to its appearance in Normal Mode.
- Expert Mode scripts containing errors cannot be exported or converted to Normal Mode; if you try to convert the script, you'll receive an error message.

To switch editing modes:

Choose Normal Mode or Expert Mode from the pop-up menu at the upper right of the Actions panel. A check mark indicates the selected mode.

To set an editing mode preference:

1. Choose Edit > Preferences.
2. Select the General tab
3. In the Actions Panel section, select Normal Mode or Expert Mode from the pop-up menu.

Using an external editor

Although the Actions panel's Expert Mode gives you more control when editing ActionScript, you can also choose to edit a script outside Flash. You can then use the `include` action to add the scripts you wrote in the external editor to a script within Flash.

For example, the following statement imports a script file:

```
#include "externalfile.as"
```

The text of the script file replaces the `include` action. The text file must be present when the movie is exported.

To add the scripts written in an external editor to a script within Flash:

1 Drag `include` from the Toolbox list to the Script window.

2 Enter the path to the external file in the Path box.

The path should be relative to the FLA file. For example, if myMovie.fla and externalfile.as were in the same folder, the path would be externalfile.as. If externalfile.as was in a subfolder called Scripts, the path would be scripts/externalfile.as.

Choosing Actions panel options

The Actions panel allows you to work with scripts in a variety of ways. You can change the font size in the Script window. You can import a text file containing ActionScript into the Actions panel and export actions as a text file, search and replace text in a script, and use syntax highlighting to make scripts easier to read and errors easier to detect. The Actions panel displays warning highlights for syntax errors and Flash Player version incompatibilities. It also highlights *deprecated*, or no longer preferable, ActionScript elements.

These Actions panel options are available in both Normal and Expert Modes unless otherwise noted.

To change the font size in the Script window:

1 From the pop-up menu at the upper right of the Actions panel, choose Font Size.

2 Select Small, Normal, or Large.

To import a text file containing ActionScript:

1 From the pop-up menu at the upper right of the Actions panel, choose Import from File.

2 Select a text file containing ActionScript, and click Open.

Note: Scripts with syntax errors can only be imported in Expert Mode. In Normal Mode, you'll receive an error message.

To export actions as a text file:

1 From the pop-up menu at the upper right of the Actions panel, Choose Export as File.

2 Choose a location where the file will be saved, and click Save.

To print actions:

1 From the pop-up menu at the upper right of the Actions panel, choose Print.

The Print dialog box appears.

2 Choose Options and click Print.

Note: The printed file will not include information about its originating Flash file. It's a good idea to include this information in a `comment` action in the script.

To search for text in a script, choose an option from the Actions panel pop-up menu:

- Choose Goto Line to go to a specific line in a script.
- Choose Find to find text.
- Choose Find Again to find text again.
- Choose Replace to find and replace text.

  In Expert Mode, Replace scans the entire body of text in a script. In Normal Mode, Replace searches and replaces text only in the parameter field of each action. For example, you cannot replace all `gotoAndPlay` actions with `gotoAndStop` in Normal Mode.

Note: Use the Find or Replace command to search the current Actions list. To search through text in every script in a movie, use the Movie Explorer. For more information, see *Using Flash*.

Highlighting and checking syntax

Syntax highlighting identifies certain ActionScript elements with specific colors. This helps prevent syntax errors such as incorrect capitalization of keywords. For example, if the keyword `typeof` was spelled `typeOf`, it would not be blue and you could recognize the error. When syntax highlighting is turned on, text is highlighted in the following way:

- Keywords and predefined identifiers (for example, `gotoAndStop`, `play`, and `stop`) are blue.
- Properties are green.
- Comments are magenta.
- Strings surrounded by quotation marks are gray.

To turn syntax highlighting on or off:

Choose Colored Syntax from the pop-up menu at the upper right of the Actions panel. A check mark indicates that the option is turned on. All scripts in your movie will be highlighted.

It's a good idea to check a script's syntax for errors before exporting a movie. Errors are reported in the Output window. You can export a movie that contains erroneous scripts. However, you will be warned that scripts containing errors were not exported.

To check the current script's syntax for errors:

Choose Check Syntax from the pop-up menu at the upper right of the Actions panel.

About error highlighting

All syntax errors are highlighted with a solid red background in the Script window in Normal Mode. This makes it easy to spot problems. If you move the mouse pointer over an action with incorrect syntax, a tooltip displays the error message associated with that action. When you select the action, the error message is also displayed in the pane title of the parameters area.

In Normal Mode all ActionScript export incompatibilities are highlighted with a solid yellow background in the Script window. For example, if the Flash Player export version is set to Flash 4, ActionScript that is supported only by the Flash 5 Player is highlighted in yellow. The export version is determined in the Publish Settings dialog box.

All deprecated actions are highlighted with a green background in the toolbox. Deprecated actions are only highlighted when the Flash export version is set to Flash 5.

To set the Flash Player export version:

1. Choose File > Publish Settings.
2. Click the Flash tab.
3. Choose an export version from the Version pop-up menu.

Note: You cannot turn off syntax error highlighting.

To show deprecated syntax highlighting:

Choose Show Deprecated Syntax from the Actions panel pop-up menu.

For a complete list of all error messages, see Appendix C, “Error Messages.”

Assigning actions to objects

You can assign an action to a button or a movie clip to make an action execute when the user clicks a button or rolls the pointer over it, or when the movie clip loads or reaches a certain frame. You assign the action to an instance of the button or movie clip; other instances of the symbol aren't affected. (To assign an action to a frame, see "Assigning actions to frames" on page 35.)

When you assign an action to a button, you must nest the action inside an `on(mouse event)` handler and specify the mouse or keyboard events that trigger the action. When you assign an action to a button in Normal Mode, the `on(mouse event)` handler is automatically inserted.

When you assign an action to a movie clip, you must nest the action inside an `onClipEvent` handler and specify the clip event that triggers the action. When you assign an action to a movie clip in Normal Mode, the `on(mouse event)` handler is automatically inserted.

The following instructions describe how to assign actions to objects using the Actions panel in Normal Mode.

Once you've assigned an action, use the Control > Test Movie command to test whether it works. Most actions won't work in Editing Mode.

To assign an action to a button or movie clip:

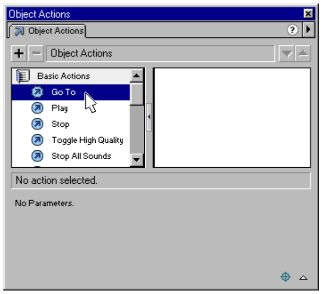
1 Select a button or movie clip instance and choose Window > Actions.

If the selection is not a button, a movie clip instance, or a frame, or if the selection includes multiple objects, the Actions panel is dimmed.

2 Choose Normal Mode from the pop-up menu at the upper right of the Object Actions panel.

3 To assign an action, do one of the following:

- Click the Actions folder in the Toolbox list on the left side of the Actions panel. Double-click an action to add it to the Actions list on the right side of the panel.
- Drag an action from the Toolbox list to the Actions list.
- Click the Add (+) button and choose an action from the pop-up menu.
- Use the keyboard shortcut listed next to each action in the pop-up menu.



Selecting an object from the toolbox in Normal Mode

4 In the Parameters fields at the bottom of the panel, select parameters for the action as needed.

Parameters vary depending on the action you choose. For detailed information on the required parameters for each action, see Chapter 7, "ActionScript Dictionary." To insert a Target path for a movie clip into a Parameter field, click the Target Path button in the lower right corner of the Actions panel. For more information, see Chapter 4, "Working with Movie Clips."

5 Repeat steps 3 and 4 to assign additional actions as necessary.

To test an object action:

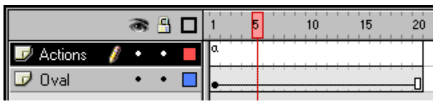
Choose Control > Test Movie.

Assigning actions to frames

To make a movie do something when it reaches a keyframe, you assign a frame action to the keyframe. For example, to create a loop in the Timeline between frames 20 and 10, you would add the following frame action to frame 20:

```
gotoAndPlay (10);
```

It's a good idea to place frame actions in a separate layer. Frames with actions display a small *a* in the Timeline.



An "a" in a keyframe indicates a frame action.

Once you've assigned an action, choose Control > Test Movie to test whether it works. Most actions won't work in Editing Mode.

The following instructions describe how to assign frame actions using the Actions panel in Normal Mode. (For information on assigning an action to a button or movie clip, see "Assigning an action or method" on page 114.)

To assign an action to a keyframe:

1 Select a keyframe in the Timeline and choose Window > Actions.

If a selected frame is not a keyframe, the action is assigned to the previous keyframe. If the selection is not a frame, or if the selection includes multiple keyframes, the Actions panel is dimmed.

2 Choose Normal Mode from the pop-up menu at the upper right of the Frame Actions panel.

3 To assign an action, do one of the following:

- Click the Actions folder in the Toolbox list on the left side of the Actions panel. Double-click an action to add it to the Actions list on the right side of the panel.
- Drag an action from the Toolbox list to the Actions list.
- Click the Add (+) button and choose an action from the pop-up menu.
- Use the keyboard shortcut listed next to each action in the pop-up menu.
- In the Parameters fields at the bottom of the panel, select parameters for the action as needed.

4 To assign additional actions, select another keyframe and repeat step 3.

To test a frame action:

Choose Control > Test Movie.

2

CHAPTER 2
Writing Scripts with ActionScript

When you create scripts in ActionScript, you can choose the level of detail you want to use. To use simple actions, you can use the Actions panel in Normal Mode and build scripts by choosing options from menus and lists. However, if you want to use ActionScript to write more powerful scripts, you must understand how ActionScript works as a language.

Like other scripting languages, ActionScript consists of components, such as predefined objects and functions, and it allows you to create your own objects and functions. ActionScript follows its own rules of syntax, reserves keywords, provides operators, and allows you to use variables to store and retrieve information.

ActionScript's syntax and style closely resemble that of JavaScript. Flash 5 performs conversions on ActionScript written in any previous version of Flash.

Using ActionScript's syntax

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 following are general rules that apply to all ActionScript. Most ActionScript terms also have their own individual requirements; for the rules for a specific term, see the its entry in Chapter 7, "ActionScript Dictionary."

Dot syntax

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

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

As another example, `submit` is a variable set in the movie clip `form` 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 `play` method of the `ballMC` instance moves the playhead in the Timeline of `ballMC`, as in the following statement:

```
ballMC.play();
```

Dot syntax also uses two special aliases, `_root` and `_parent`. 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 movie clip is nested. You can use `_parent` to create a relative target path. For example, if the movie clip `dog` is nested inside the movie clip `animal`, the following statement on the instance `dog` tells `animal` to stop:

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

See Chapter 4, "Working with Movie Clips."

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 the Flash 5 Player, but its use is not recommended. In slash syntax, slashes are used instead of dots to indicate the path to a movie clip or variable. To indicate a variable, you precede the variable with a colon as in the following:

```
myMovieClip/childMovieClip:myVariable
```

You can write the same target path in dot syntax, as in the following:

```
myMovieClip.childMovieClip.myVariable
```

Slash syntax was most commonly used with the `tellTarget` action, whose use is also no longer recommended.

Note: The `with` action is now preferred over `tellTarget` because it is more compatible with dot syntax. For more information, see their individual entries in Chapter 7, "ActionScript Dictionary."

Curly braces

ActionScript statements are grouped together into blocks with curly braces (`{ }`), as in the following script:

```
on(release) {
   myDate = new Date();
   currentMonth = myDate.getMonth();
}
```

See "Using actions" on page 58.

Semicolons

An ActionScript statement is terminated with a semicolon, but if you omit the terminating semicolon, Flash will still compile your script successfully. For example, the following statements are terminated with semicolons:

```
column = passedDate.getDay();
row    = 0;
```

The same statements could be written without the terminating semicolons:

```
column = passedDate.getDay()
row    = 0
```

Parentheses

When you define a function, place any arguments inside parentheses:

```
function myFunction (name, age, reader){
   ...
}
```

When you call a function, include any arguments passed to the function in parentheses, as shown here:

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

You can also use parentheses to override ActionScript's order of precedence or to make your ActionScript statements easier to read. See "Operator precedence" on page 52.

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 new color object:

```
onClipEvent(enterFrame) {
   (new Color(this)).setRGB(0xffffff));
}
```

If you didn't use parentheses, you would need to add a statement to the code to evaluate it:

```
onClipEvent(enterFrame) {
   myColor = new Color(this);
   myColor.setRGB(0xffffff);
}
```

Uppercase and lowercase letters

Only keywords in ActionScript are case sensitive; with the rest of ActionScript, you can use uppercase and lowercase letters however you want. For example, the following statements are equivalent:

```
cat.hilite = true;
CAT.hilite = true;
```

However, it's a good habit to follow consistent capitalization conventions, such as those used in this book, to make it is easier to identify names of functions and variables when reading ActionScript code.

If you don't use correct capitalization with keywords, your script will have errors. When Colored Syntax is turned on in the Actions panel, keywords written with the correct capitalization are blue. For more information, see "Keywords" on page 41 and "Highlighting and checking syntax" on page 31.

Comments

In the Actions panel, use the `comment` statement to add notes to a frame or button action when you want to keep track of what you intended an action to do. Comments are also useful for passing information to other developers if you work in a collaborative environment or are providing samples.

When you choose the `comment` action, the characters `//` are inserted into the script. Even a simple script is easier to understand if you make notes as you create it:

```
on(release) {
   // create new Date object
   myDate = new Date();
   currentMonth = myDate.getMonth();
   // convert month number to month name
   monthName = calcMonth(currentMonth);
   year = myDate.getFullYear();
   currentDate = myDate.getDat ();
}
```

Comments appear in pink in the Script window. They 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.

Keywords

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

`break`	`for`	`new`	`var`
`continue`	`function`	`return`	`void`
`delete`	`if`	`this`	`while`
`else`	`in`	`typeof`	`with`

For more information about a specific keyword, see its entry in Chapter 7, "ActionScript Dictionary."

Constants

A constant is a property whose value never changes. Constants are listed in the Actions toolbox and in Chapter 7, "ActionScript Dictionary," in all uppercase letters.

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

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

About data types

A data type describes the kind of information a variable or ActionScript element can hold. There are two kinds of data types: primitive and reference. The primitive data types—string, number, and Boolean—have a constant value and, therefore, can hold the actual value of the element they represent. The reference data types—movie clip and object—have values that can change and, therefore, contain references to the actual value of the element. Variables containing primitive data types behave differently in certain situations than those containing reference types. See "Using variables in a script" on page 49.

Each data type has its own rules and is listed here. References are included for data types that are discussed in more detail.

String

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. Strings are treated as characters instead of as variables. For example, in the following statement, `"L7"` is a string:

```
favoriteBand = "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:

```
greeting = "Welcome," + firstName;
```

Although ActionScript does not distinguish between uppercase and lowercase in references to variables, instance names, and frame labels, literal strings are case sensitive. For example, the following two statements place different text into the specified text field variables, because `"Hello"` and `"HELLO"` are literal strings.

```
invoice.display = "Hello";
invoice.display = "HELLO";
```

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

Number

The number data type is a double-precision floating-point number. You can manipulate numbers using the arithmetic operators addition (+), subtraction (-), multiplication (*), division (/), modulo (%), increment (++), and decrement (--). You can also use methods of the predefined Math object to manipulate numbers. The following example uses the `sqrt` (square root) method to return the square root of the number 100:

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

See “Numeric operators” on page 53.

Boolean

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. For example, in the following script, the movie plays if the variable `password` is `true`:

```
onClipEvent(enterFrame) {
   if ((userName == true) && (password == true)){
      play();
   }
}
```

See “Using "if" statements” on page 60 and “Logical operators” on page 54.

Object

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 allows you to 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
```

You can use ActionScript's predefined objects to access and manipulate specific kinds of information. For example, the Math object has methods that perform mathematical operations on numbers you pass to them. This example uses the `sqrt` method:

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

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 your own objects so that you can organize information in your movie. To add interactivity to a movie with ActionScript, you'll need many different pieces of information: for example, you might need a user's name, the speed of a ball, the names of items in a shopping cart, the number of frames loaded, the user's zip code, and which key was pressed last. Creating custom objects allows you to organize this information into groups, simplify your scripting, and reuse your scripts. For more information, see "Using custom objects" on page 72.

Movie clip

Movie clips are symbols that can play animation in a Flash movie. They are the only data type that refers to a graphical element. The movie clip data type allows you to control movie clip symbols using the methods of the MovieClip object. You call the methods using the dot (.) operator, as shown here:

```
myClip.startDrag(true);
parentClip.childClip.getURL( "http://www.macromedia.com/support/"
+ product);
```

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 movie plays, you can record and save information about what the user has done, record values that change as the movie plays, or evaluate whether some 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 movie. Initializing variables makes it easier to track and compare the variable's value as the movie plays.

Variables can hold any type of data: number, string, Boolean, object, or movie clip. 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 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 movie and movie clip instance has its own set of variables, with each variable having its own value independent of variables in other movies or movie clips.

Naming a variable

A variable's name must follow these rules:

- It must be an identifier.
- It cannot be a keyword or a Boolean literal (`true` or `false`).
- It must be unique within its scope. (See "Scoping a variable" on page 48.)

Typing a variable

In Flash, you do not have to explicitly define a variable as holding either a number, a string, or other data type. Flash determines the data type of a variable when the variable is assigned:

```
x = 3;
```

In the expression `x = 3`, Flash evaluates the element on the right side of the operator and determines that it is of type number. A later assignment may change the type of `x`; for example, `x = "hello"` changes the type of `x` to a string. A variable that hasn't been assigned a value has a type of `undefined`.

ActionScript converts data types automatically when an expression requires it. For example, when you pass a value to the `trace` action, `trace` automatically converts the value to a string and sends it to the Output window. In expressions with operators, ActionScript converts data types as needed; for example, when used with a string, the + 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"
```

When you debug scripts, it's often useful 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 `typeof` operator, as in this example:

```
trace(typeof(variableName));
```

To convert a string to a numerical value, use the `Number` function. To convert a numerical value to a string, use the `String` function. See their individual entries in Chapter 7, "ActionScript Dictionary" on page 157.

Scoping a variable

A variable's "scope" refers to the area in which the variable is known and can be referenced. Variables in ActionScript can be either global or local. A global variable is shared among all Timelines; a local variable is only available within its own block of code (between the curly braces).

You can use the `var` statement to declare a local variable inside a script. 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 only exists inside the function `makeDays`:

```
function makeDays(){
   var i
   for( i = 0; i < monthArray[month]; i++ ) {

      _root.Days.attachMovie( "DayDisplay", i, i + 2000 );

      _root.Days[i].num = i + 1;
      _root.Days[i]._x = column * _root.Days[i]._width;
      _root.Days[i]._y = row * _root.Days[i]._height;

      column = column + 1;

      if (column == 7 ) {

         column = 0;
         row = row + 1;
      }
   }
}
```

Local variables can also help prevent name collisions, which can cause errors in your movie. For example, if you use `name` as a local variable, you could use it to store a user name in one context and a movie clip instance name in another; because these variables would run in separate scopes, there would be no collision.

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 only changeable within its own block of code. If an expression in a function uses a global variable, something outside the function could change its value, which would change the function.

Variable declaration

To declare global variables, use the `setVariables` action or the assignment (=) operator. Both methods achieve the same results.

To declare local variables, use the `var` statement inside the body of a function. Local variables are scoped to the block, and expire at the end of the block. Local variables not declared within a block expire at the end of their script.

Note: The `call` action also creates a new local variable scope for the script it calls. When the called script exits, this local variable scope disappears. However, this is not recommended because the `call` action has been replaced by the `with` action which is more compatible with dot syntax.

To test the value of a variable, use the `trace` action to send the value to the Output window. For example, `trace(hoursWorked)` sends the value of the variable `hoursWorked` to the Output window in test-movie mode. You can also check and set the variable values in the Debugger in test-movie mode. For more information, see Chapter 6, "Troubleshooting ActionScript."

Using variables in a script

You must declare a variable in a script before you can use it in an expression. If you use an undeclared variable, as in the following example, the variable's value will be `undefined` and your script will generate an error:

```
getURL(myWebSite);
myWebSite = "http://www.shrimpmeat.net";
```

The statement declaring the variable `myWebSite` must come first so that the variable in the `getURL` action can be replaced with a value.

You can change the value of a variable many times in a script. The type of data that the variable contains affects how and when the variable changes. Primitive data types, such as strings and numbers, are passed by value. This means that the actual content of the variable is passed to the variable.

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

```
var x = 15;
var y = x;
var x = 30;
```

As another example, the variable `in` contains a primitive value, 9, so the actual value is passed to the `sqrt` function and the returned value is 3:

```
function sqrt(x){
   return x * x;
}

var in = 9;
var out = sqr(in);
```

The value of the variable `in` does not change.

The object data type can contain such a large and complex amount of information that a variable with this type doesn't hold the actual value; it holds a reference to the value. This reference is like 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 is an example of passing by reference:

```
var myArray = ["tom", "dick"];
var newArray = myArray;
myArray[1] = "jack";
trace(newArray);
```

The above code creates an Array object called `myArray` that has two elements. The variable `newArray` is created and passed a reference to `myArray`. When the second element of `myArray` is changed, it affects every variable with a reference to it. The `trace` action would send `["tom", "jack"]` to the Output window.

In the next example, `myArray` contains an Array object, so it is passed to function `zeroArray` by reference. The `zeroArray` function changes the content of the array in `myArray`.

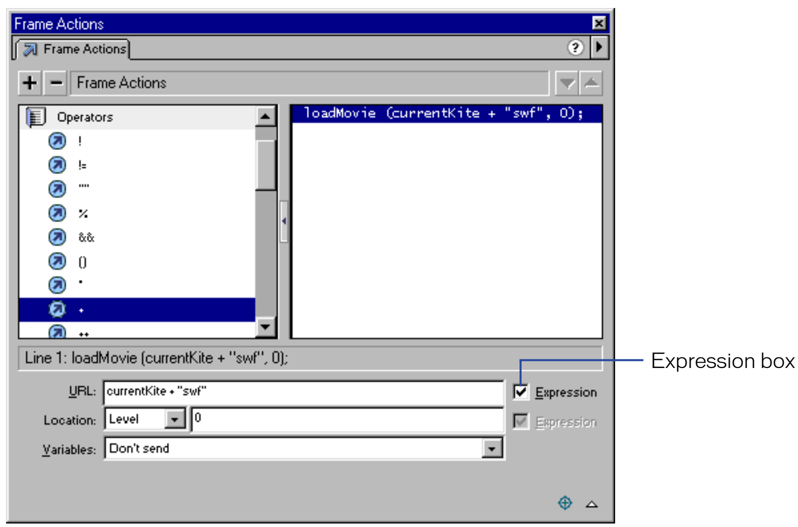
```
function zeroArray (array){
   var i;
   for (i=0; i < array.length; i++) {
      array[i] = 0;
   }
}

var myArray = new Array();
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;

var out = zeroArray(myArray)
```

The function `zeroArray` accepts an Array object as an argument and sets all the elements of that array to 0. It can modify the array because the array is passed by reference.

References to all objects other than movie clips are called *hard references* because if an object is referenced, it cannot be deleted. A reference to a movie clip is a special kind of reference called a *soft reference*. Soft references do not force the referenced object to exist. If a movie clip is destroyed with an action such as `removeMovieClip`, any reference to it will no longer work.

Using operators to manipulate values in expressions

An expression is any statement that Flash can evaluate that will return a value. You can create an expression by combining operators and values, or by calling a function. When you write an expression in the Actions panel in Normal Mode, make sure the Expression box is checked in the Parameters panel, otherwise the field will contain the literal value of a string.



Check the Expression box next to a field to create an expression

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 + 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. For detailed information on each operator mentioned here, as well as special operators that don't fall into these categories, see Chapter 7, "ActionScript Dictionary."

Operator precedence

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.

For a table of all operators and their precedence, see Appendix B, "Operator Precedence and Associativity."

Operator associativity

When two or more operators share the same precedence, their associativity determines the order in which they are performed. Associativity can either be 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;
```

For a table of all operators and their associativity, see Appendix B, "Operator Precedence and Associativity."

Numeric operators

Numeric operators add, subtract, multiply, divide, and perform other arithmetic operations. Parentheses and the minus sign are arithmetic operators. The following table lists ActionScript's numeric operators:

Operator	Operation performed
+	Addition
*	Multiplication
/	Division
%	Modulo
-	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 variable `score` is `100`, a certain movie loads; otherwise, a different movie loads:

```
if (score == 100){
   loadMovie("winner.swf", 5);
} else {
   loadMovie("loser.swf", 5);
   }
```

The following table lists ActionScript's comparison operators:

Operator	Operation performed
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

String operators

The + 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 + operator's operands is a string, Flash converts the other operand to a 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 only compare strings 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.

Note: ActionScript's data typing in Flash 5 allows the same operators to be used on different types of data. It is no longer necessary to use the Flash 4 string operators (for example, `eq`, `ge`, and `lt`) unless you are exporting as a Flash 4 movie.

Logical operators

Logical operators compare Boolean (`true` and `false`) values 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 `false`. Logical operators are often used in conjunction with comparison operators to determine the condition of an `if` action. For example, in the following script, if both expressions are `true`, the `if` action will execute:

```
if ((i > 10) && (_framesloaded > 50)){
   play();
}
```

The following table lists ActionScript's logical operators:

Operator	Operation performed
&&	Logical AND
||	Logical OR
!	Logical NOT

Bitwise operators

Bitwise operators internally manipulate floating-point numbers to change them into 32-bit integers, which are easier to work with. The exact bitwise operation performed depends on the operator, but all bitwise operations evaluate each digit of a floating-point number separately to compute a new value.

The following table lists ActionScript's bitwise operators:

Operator	Operation performed
&	Bitwise And
|	Bitwise Or
^	Bitwise Xor
~	Bitwise Not
<<	Shift left
>>	Shift right
>>>	Shift right zero fill

Equality and assignment operators

You can use the equality (==) operator to determine whether the values or identities 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.

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

```
password = "Sk8tEr";
```

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

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

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

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

The following table lists ActionScript's equality and assignment operators:

Operator	Operation performed
==	Equality
!=	Inequality
=	Assignment
+=	Addition and assignment
-=	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 any predefined or custom ActionScript object properties, including those of a movie clip.

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 are examples using 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 and the array access operator evaluates its contents to a name and then accesses the value of that named property. For example, the following two lines of code access the same variable `velocity` in the movie clip `rocket`:

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

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 `[]` 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 ]
```

If you are familiar with the Flash 4 ActionScript slash syntax, you may have done the same thing using the `eval` function, as in the following:

```
eval("mc" & i);
```

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

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

Again, this is equivalent to the following Flash 4 ActionScript slash syntax:

```
Set Variable: "name:" & index = "Gary"
```

The array access operator can also be nested with itself to simulate multidimensional arrays.

```
chessboard[row][column]
```

This is equivalent to the following slash syntax:

```
eval("chessboard/" & row & ":" & column)
```

Note: If you want to write ActionScript that is compatible with the Flash 4 Player, you can use the `eval` action with the `add` operator.

Using actions

Actions are ActionScript's statements, or commands. Multiple actions assigned to the same frame or object create a script. Actions can act independently of each other, as in the following statements:

```
swapDepths("mc1", "mc2");
gotoAndPlay(15);
```

You can also nest actions by using one action inside another; this allows actions to affect each other. In the following example, the `if` action tells the `gotoAndPlay` action when to execute:

```
if (i >= 25) {
   gotoAndPlay(10);
}
```

Actions can move the playhead in the Timeline (`gotoAndPlay`), control the flow of a script by creating loops (`do while`) or conditional logic (`if`), or create new functions and variables (`function`, `setVariable`). The following table lists all ActionScript actions:

Actions				
break	evaluate	include	print	stopDrag
call	for	loadMovie	printAsBitmap	swapDepths
comment	for...in	loadVariables	removeMovieClip	tellTarget
continue	fsCommand	nextFrame nextScene	return	toggleHighQuality
delete	function	on	setVariable	stopDrag
do...while	getURL	onClipEvent	setProperty	trace
duplicate MovieClip	gotoAndPlay gotoAndStop	play	startDrag	unloadMovie
else	if	prevFrame	stop	var
else if	ifFrameLoaded	prevScene	stopAllSounds	while

For syntax and usage examples of each action, see individual entries in Chapter 7, "ActionScript Dictionary."

Note: In this book, the ActionScript term *`action`* is synonymous with the JavaScript term *`statement`*.

Writing a target path

To use an action to control a movie clip or loaded movie, you must specify its name and its address, called a *target path*. The following actions take one or more target paths as arguments:

- `loadMovie`
- `loadVariables`
- `unloadMovie`
- `setProperty`
- `startDrag`
- `duplicateMovieClip`
- `removeMovieClip`
- `print`
- `printAsBitmap`
- `tellTarget`

For example, the `loadMovie` action takes the arguments *`URL`*, *`Location`*, and *`Variables`*. The *`URL`* is the location on the Web of the movie you want to load. The *`Location`* is the target path into which the movie will be loaded.

```
loadMovie(URL, Location, Variables);
```

Note: The *`Variables`* argument is not required for this example.

The following statement loads the URL `http://www.mySite.com/myMovie.swf` into the instance `bar` on the main Timeline, `_root`; `_root.bar` is the target path;

```
loadMovie("http://www.mySite.com/myMovie.swf", _root.bar);
```

In ActionScript you identify a movie clip by its instance name. 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 an instance name:

1. Select the movie clip on the Stage.
2. Choose Window > Panels > Instance.
3. Enter an instance name in the Name field.

To identify a loaded movie:

Use `_levelX` where `X` is the level number specified in the `loadMovie` action that loaded the movie.

For example, a movie loaded into level 5 has the instance name `_level5`. In the following example, a movie is loaded into level 5 and its visibility is set to false:

```
onClipEvent(load) {
   loadMovie("myMovie.swf", 5);
}
onClipEvent(enterFrame) {
   _level5._visible = false;
}
```

To enter a movie's target path:

Click the Insert Target Path button in the Actions panel, and select a movie clip from the list that appears.

For more information about writing target paths, see Chapter 4, "Working with Movie Clips."

Controlling flow in scripts

ActionScript uses `if`, `for`, `while`, `do...while`, and `for...in` actions to perform an action depending on whether a condition exists.

Using "if" statements

Statements that check whether a condition is `true` or `false` begin with the term `if`. If the condition exists, ActionScript executes the statement that follows. 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 several conditions. The term `else if` specifies alternative tests to perform if previous conditions are false.

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

Repeating an action

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) {
   myMC.duplicateMovieClip("newMC" + i, i );
   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, as in the following:

```
i = 4
do {
   myMC.duplicateMovieClip("newMC" +i, i );
   i --;
} while (i > 0);
```

To repeat an action using a built-in counter:

Use the `for` statement.

Most loops use a counter of some kind to control how many times the loop runs. 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, as in the following:

```
for (i = 4; i > 0; i--){
   myMC.duplicateMovieClip("newMC" + i, i + 10);
}
```

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

Use the `for..in` statement.

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

```
myObject = { 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 window:

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

```
for (name in myMovieClip) {
   if (typeof (myMovieClip[name]) == "movieclip") {
      trace("I have a movie clip child named " + name);
   }
}
```

Note: The `for..in` statement iterates over properties of objects in the iterated object's prototype chain. If a child object's prototype is `parent`, `for..in` will also iterate over the properties of `parent`. See "Creating inheritance" on page 74.

For more information on each action, see individual entries in Chapter 7, "ActionScript Dictionary."

Using predefined functions

A function is a block of ActionScript code that can be reused anywhere in a movie. If you pass specific values called arguments to a function, the function will operate on those values. A function can also return values. Flash has predefined functions that allow you to access certain information and perform certain tasks, such as collision detection (`hitTest`), getting the value of the last key pressed (`keycode`), and getting the version number of the Flash Player hosting the movie (`getVersion`).

Calling a function

You can call a function in any Timeline from any Timeline, including a loaded movie. Each function has its own characteristics and some require you to pass certain values. If you pass more arguments than the function requires, the extra values are ignored. If you don't pass a required argument, the empty arguments are assigned the `undefined` data type, which can cause errors when you export a script. To call a function, it must be in a frame that the playhead has reached.

Flash's predefined functions are listed in the following table:

Boolean	getTimer	isFinite	newline	scroll
escape	getVersion	isNaN	number	String
eval	globalToLocal	keycode	parseFloat	targetPath
false	hitTest	localToGlobal	parseInt	true
getProperty	int	maxscroll	random	unescape

Note: String functions are deprecated and are not listed in the above table.

To call a function in Expert Mode:

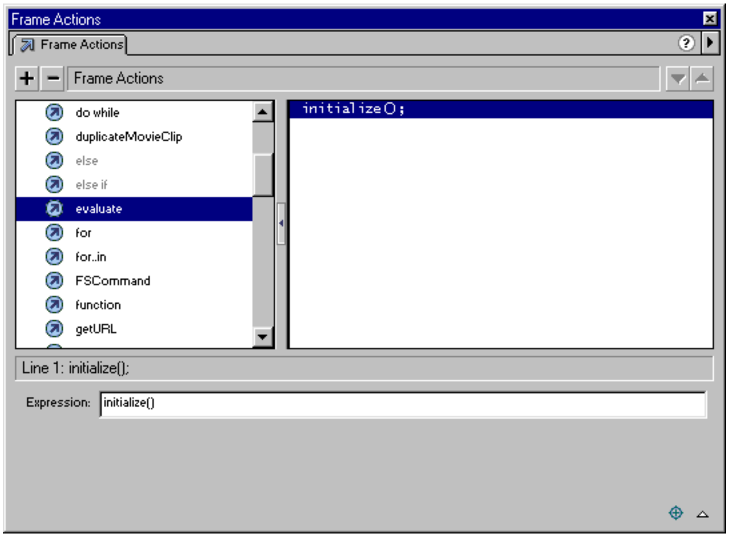
Use the name of the function. Pass any required arguments inside parentheses.

The following example calls the `initialize` function which requires no arguments:

```
initialize();
```

To call a function in Normal Mode:

Use the `evaluate` action. Enter the function name and any required arguments in the Expression field.



Use the evaluate action to call a function in Normal Mode

To call a function on another Timeline use a target path. For example, to call the function `calculateTax` that was declared in the instance `functionsMovieClip`, use the following path:

```
_root.functionsMovieClip.calculateTax(total);
```

Note: Pass any arguments inside the parentheses.

For more information on each function, including deprecated string functions, see individual entries in Chapter 7, "ActionScript Dictionary."

Creating custom functions

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

A function can be thought of as a "black box": when a function is called, it is provided with input (arguments). It performs some operation and then generates output (a return value). A well-written function has carefully placed comments about its input, output, and purpose. This way, a user of the function does not need to understand exactly how the function works.

Defining a function

Functions, like variables, are attached to the movie clip that defines them. When a function is redefined, the new definition replaces the old definition.

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

The following is a function named `Circle` with the argument `radius`:

```
function Circle(radius) {
   this.radius = radius;
   this.area = Math.PI * radius * radius;
}
```

Note: The keyword `this`, used in a function body, is a reference to the movie clip that the function belongs to.

You can also define a function by creating a *function literal*. A function literal is 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 in the following:

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

Passing arguments to a function

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

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

When the function is called, the required arguments must be passed to the function. The function substitutes the passed values for the arguments in the function definition. In this example, `scorecard` is the instance name of a movie clip; `display` and `score` are input text fields in the instance. The following function call assigns the variable `display` the value `"JEB"` and the variable `score` the value `45000`:

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

The argument `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 arguments during a function call, the omitted arguments are passed as `undefined`. If you provide extra arguments in a function call that are not required by the function declaration, they are ignored.

Using local variables in a function

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

Note: If you modify global variables in a function, use script comments to document these modifications.

Returning values from a function

You can use the `return` action to return values from functions. The `return` action stops the function and replaces it with the value of the `return` action. If Flash doesn't encounter a `return` action before the end of a function, an empty string is returned. For example, the following function returns the square of the argument `x`:

```
function sqr(x) {
   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 = _root.boat._x;
   boat_y = _root.boat._y;
   car_x = _root.car._x;
   car_y = _root.car._y;
}
```

Calling a function

To invoke a function using the Actions panel in Normal Mode, you use the `evaluate` action. Pass the required arguments inside parentheses. You can call a function in any Timeline from any Timeline, including a loaded movie. For example, the following statement invokes the function `sqr` in movie clip `MathLib` on the main Timeline, passes it the argument 3, and stores the result in the variable `temp`:

```
var temp = _root.MathLib.sqr(3);
```

In Flash 4, to simulate calling a function you could write a script on a frame after the end of the movie and invoke it by passing the name of the frame label to the `call` action. For example, if a script that initialized variables was on a frame labeled `initialize`, you would call it as follows:

```
call("initialize");
```

This kind of script was not a true function because it could not accept arguments and it could not return a value. Although the `call` action still functions in Flash 5, its use is not recommended.

Using predefined objects

You can use Flash's predefined objects to access certain kinds of information. Most predefined objects have *methods* (functions assigned to an object) that you can call to return a value or perform an action. For example, the Date object returns information from the system clock and the Sound object lets you control sound elements in your movie.

Some predefined objects have properties whose values you can read. For example, the Key object has constant values that represent keys on the keyboard. Each object has its own characteristics and abilities that can be used in your movie.

The following are Flash's predefined objects:

- Array
- Boolean
- Color
- Date
- Key
- Math
- MovieClip
- Number
- Object
- Selection
- Sound
- String
- XML
- XMLSocket

Movie clip instances are represented as objects in ActionScript. You can call predefined movie clip methods just as you would call the methods of any other ActionScript object.

For detailed information on each object, see its entry in Chapter 7, "ActionScript Dictionary."

Creating an object

There are two ways to create an object: the `new` operator and the object initializer operator (`{}`). You can use the `new` operator to create an object from a predefined object class, or from a custom defined object class. You can use the object initializer operator (`{}`) to create an object of generic type Object.

To use the `new` operator to create an object, you need to use it with a constructor function. (A constructor function is simply a function whose sole purpose is to create a certain type of object.) ActionScript's predefined objects are essentially prewritten constructor functions. The new object *instantiates,* or creates, a copy of the object and assigns it all the properties and methods of that object. This is similar to dragging a movie clip from the Library to the Stage in a movie. For example, the following statements instantiate a Date object:

```
currentDate = new Date();
```

You can access the methods of some predefined objects without instantiating them. For example, the following statement calls the Math object method `random`:

```
Math.random();
```

Each object that requires a constructor function has a corresponding element in the Actions panel toolbox; for example, `new Color`, `new Date`, `new String`, and so on.

To create an object with the `new` operator in Normal Mode:

1 Choose `setVariable`

2 Enter an identifier in the Name field.

3 Enter `new Object`, `new Color`, and so on in the Value field. Enter any arguments required by the constructor function in parentheses.

4 Check the Expression box of the Value field.

   If you don't check the Expression box, the entire value will be a string literal.

In the following code, the object `c` is created from the constructor Color:

```
c = new Color(this);
```

Note: An object name is a variable with the object data type assigned to it.

To access a method in Normal Mode:

1 Select the `evaluate` action.

2 Enter the name of the object in the Expression field.

3 Enter a property of the object in the Expression field.

To use the object initializer operator (`{}`) in Normal Mode:

1. Select the `setVariable` action.
2. Enter name in the Variable field; this is the name of the new object.
3. Enter the property name and value pairs separated by a colon inside the object initializer operator (`{}`).

For example, in this statement the property names are `radius` and `area` and their values are `5` and the value of an expression:

```
myCircle = {radius: 5, area:(pi * radius * radius)};
```

The parentheses cause the expression to evaluate. The returned value is the value of the variable `area`.

You can also nest array and object initializers, as in this statement:

```
newObject = {name: "John Smith", projects: ["Flash",
"Dreamweaver"]};
```

For detailed information on each object, see its entry in Chapter 7, "ActionScript Dictionary."

Accessing object properties

Use the dot (`.`) operator to access the value of properties in an object. The name of the object goes on the left side of the dot, and the name of the property goes on the right side. For example, in the following statement, `myObject` is the object and `name` is the property:

```
myObject.name
```

To assign a value to a property in Normal Mode, use the `setVariable` action:

```
myObject.name = "Allen";
```

To change the value of a property, assign a new value as shown here:

```
myObject.name = "Homer";
```

You can also use the array access operator (`[]`) to access the properties of an object. See "Dot and array access operators" on page 57.

Calling object methods

You can call an object's method by using the dot operator followed by the method. For example, the following example calls the `setVolume` method of the Sound object:

```
s = new Sound(this);
s.setVolume(50);
```

To call the method of a predefined object in Normal Mode, use the `evaluate` action.

Using the MovieClip object

You can use the methods of the predefined MovieClip object to control movie clip symbol instances on the Stage. The following example tells the instance `dateCounter` to play:

```
dateCounter.play();
```

For detailed information on the MovieClip object, see its entry in Chapter 7, "ActionScript Dictionary."

Using the Array object

The Array object is a commonly used predefined ActionScript object that stores its data in numbered properties instead of named properties. An array element's name is called an *index*. This is useful for storing and retrieving certain types of information such as lists of students or a sequence of moves in a game.

You can assign elements of the Array object just as you would the property of any object:

```
move[1] = "a2a4";
move[2] = "h7h5";
move[3] = "b1c3";
...
move[100] = "e3e4";
```

To access the second element of the array, use the expression `move[2]`.

The Array object has a predefined `length` property that is the value of the number of elements in the array. When an element of the Array object is assigned and the element's index is a positive integer such that `index >= length`, `length` is automatically updated to `index + 1`.

Using custom objects

You can create custom objects to organize information in your scripts for easier storage and access by defining an object's properties and methods. After you create a master object or "class," you can use or "instantiate" copies (that is, instances) of that object in a movie. This allows you to reuse code and conserve file size.

An object is a complex data type containing zero or more properties. 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, movie clip, or `undefined`. The following properties are of various data types:

```
customer.name = "Jane Doe";
customer.age = 30;
customer.member = true;
customer.account.currentRecord = 000609;
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 an object

You can use the `new` operator to create an object from a constructor function. A constructor function is always given the same name as the type of object it is creating. For example, a constructor that creates an account object would be called `Account`. The following statement creates a new object from the function called `MyConstructorFunction`:

```
new MyConstructorFunction (argument1, argument2, ... argumentN);
```

When `MyConstructorFunction` is called, Flash passes it the hidden argument `this`, which is a reference to the object that the `MyConstructorFunction` is creating. When you define a constructor, `this` allows you to refer to the objects that the constructor will create. For example, the following is a constructor function that creates a circle:

```
function Circle(radius) {
   this.radius = radius;
   this.area = Math.PI * radius * radius;
}
```

Constructor functions are commonly used to fill in the methods of an object.

```
function Area() {
   this.circleArea = MAth.PI * radius * radius;
}
```

To use an object in a script, you must assign it to a variable. To create a new circle object with the radius 5, use the `new` operator to create the object and assign it to the local variable `myCircle`:

```
var myCircle = new Circle(5);
```

Note: Objects have the same scope as the variable to which they are assigned. See "Scoping a variable" on page 48.

Creating inheritance

All functions have a `prototype` property that is created automatically when the function is defined. When you use a constructor function to create a new object, all the properties and methods of the constructor's `prototype` property become properties and methods of the `__proto__` property of the new object. The `prototype` property indicates the default property values for objects created with that function. Passing values using the `__proto__` and `prototype` properties is called inheritance.

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 (`object.__proto__`). If the called property is not a property of the object's `__proto__` object, ActionScript looks at `object.__proto__.__proto__`.

It's common practice to attach methods to an object by assigning them to the object's `prototype` property. The following steps describe how to define a sample method:

1 Define the constructor function `Circle`, as follows:

```
function Circle(radius) {
    this.radius = radius;
}
```

2 Define the `area` method of the `Circle` object. The `area` method will calculate the area of the circle. You can use a function literal to define the `area` method and set the area property of the circle's prototype object, as follows:

```
Circle.prototype.area = function () {
        return Math.PI * this.radius * this.radius;
    }
```

3 Create an instance of the `Circle` object, as follows:

```
var myCircle = new Circle(4);
```

4 Call the `area` method of the new `myCircle` object, as follows:

```
var myCircleArea = myCircle.area()
```

ActionScript searches the `myCircle` object for the `area` method. Since the object doesn't have an `area` method, its prototype object `Circle.prototype` is searched for the `area` method. ActionScript finds it and calls it.

You can also attach a method to an object by attaching the method to every individual instance of the object, as in this example:

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

This technique is not recommended. Using the `prototype` object is more efficient, because only one definition of `area` is necessary, and that definition is automatically copied into all instances created by the `Circle` function.

The `prototype` property is supported by Flash Player version 5 and later. For more information, see Chapter 7, "ActionScript Dictionary."

Opening Flash 4 files

ActionScript has changed considerably with the release of Flash 5. It is now an object-oriented language with multiple data types and dot syntax. Flash 4 ActionScript only had 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 Flash 5, you can use one set of operators on all data types.

When you use Flash 5 to open a file that was created in Flash 4, Flash automatically converts ActionScript expressions to make them compatible with the new Flash 5 syntax. You'll see the following data type and operator conversions in your ActionScript code:

- The = operator in Flash 4 was used for numeric equality. In Flash 5, == 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, 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 obviously numbers, so they are not enclosed in `Number`).
- In Flash 4, the escape sequence `\n` generated a carriage return character (ASCII 13). In Flash 5, 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, `&` 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 parentheses, for example, `Get Timer`, `Set Variable`, `Stop`, and `Play`. To create consistent syntax, the Flash 5 `getTimer` function and all actions now require closing parentheses. These parentheses are automatically added during the conversion.
- When the `getProperty` function is executed on a movie clip that doesn't exist, it returns the value `undefined`, not 0, in Flash 5. And `undefined == 0` is `false` in Flash 5 ActionScript. Flash fixes 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 keywords as variable names in your Flash 4 ActionScript, the syntax will return an error in Flash 5. To fix this, rename your variables in all locations. See “Keywords” on page 41.

Using Flash 5 to create Flash 4 content

If you are using Flash 5 to create content for the Flash 4 Player (by exporting as Flash 4), you won't be able to take advantage of all the new features present in Flash 5 ActionScript. However, many new ActionScript features are still available. Flash 4 ActionScript has only one basic primitive data type which is used for both numeric and string manipulation. When you author a movie for the Flash 4 Player, you need to use the deprecated string operators located in the String Operators category in the toolbox.

You can use the following Flash 5 features when you export to the Flash 4 SWF file format:

- The array and object access operator (`[]`).
- The dot operator (`.`).
- Logical operators, assignment operators, and pre-increment and post-increment/decrement operators.
- The modulo operator(`%`), all methods and properties of the Math object.

  These operators and functions are not supported natively by the Flash 4 Player. Flash 5 must export them as series approximations. This means that the results are only approximate. In addition, due to the inclusion of series approximations in the SWF file, these functions take up more room in Flash 4 SWF files than they do in Flash 5 SWF files.
- The `for`, `while`, `do..while`, `break`, and `continue` actions.
- The `print` and `printAsBitmap` actions.

The following Flash 5 features can't be used in movies exported to the Flash 4 SWF file format:

- Custom functions
- XML support
- Local variables
- Predefined objects (except Math)
- Movie clip actions
- Multiple data types
- `eval` with dot syntax (for example, `eval("_root.movieclip.variable")`)
- `return`
- `new`
- `delete`
- `typeof`
- `for..in`
- `keycode`
- `targetPath`
- `escape`
- `globalToLocal` and `localToGlobal`
- `hitTest`
- `isFinite` and `inNaN`
- `parseFloat` and `parseInt`
- `unescape`
- `_xmouse` and `_ymouse`
- `_quality`

CHAPTER 3

Creating Interaction with ActionScript

An interactive movie involves your audience. Using the keyboard, the mouse, or both, your audience can jump to different parts of movies, move objects, enter information, click buttons, and perform many other interactive operations.

You create interactive movies by setting up scripts that run when specific events occur. 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 keys on the keyboard. You use ActionScript to create scripts that tell Flash what action to perform when the event occurs.

The following basic actions are common ways to control navigation and user interaction in a movie:

- Playing and stopping movies
- Adjusting a movie's display quality
- Stopping all sounds
- Jumping to a frame or scene
- Jumping to a different URL
- Checking whether a frame is loaded
- Loading and unloading additional movies

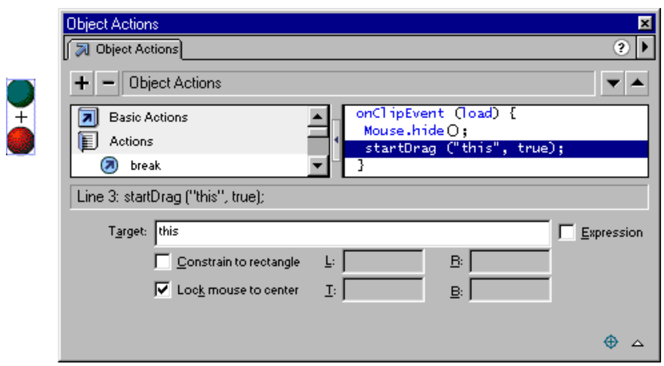
For detailed information on these actions, see *Using Flash.*

To create more complex interactivity, you need to understand the following techniques:

- Creating a custom cursor
- Getting the mouse position
- Capturing keypresses
- Creating a scrolling text field
- Setting color values
- Creating sound controls
- Detecting collisions

Creating a custom cursor

To hide the standard cursor (that is, the onscreen representation of the mouse pointer), you use the `hide` method of the predefined Mouse object. To use a movie clip as the custom cursor, you use the `startDrag` action.



Actions attached to a movie clip to create a custom cursor

To create a custom cursor:

1 Create a movie clip to use as a custom cursor.

2 Select the movie clip instance on the Stage.

3 Choose Window > Actions to open the Object Actions panel.

4 In the Toolbox list, select Objects, then select Mouse, and drag `hide` to the Script window.

   The code should look like this:

```
onClipEvent(load){
    Mouse.hide();
}
```

5 In the Toolbox list, select Actions; then drag `startDrag` to the Script window.

6 Select the Lock Mouse to Center box.

   The code should look like this:

```
onClipEvent(load){
    Mouse.hide()
    startDrag("this", true);
}
```

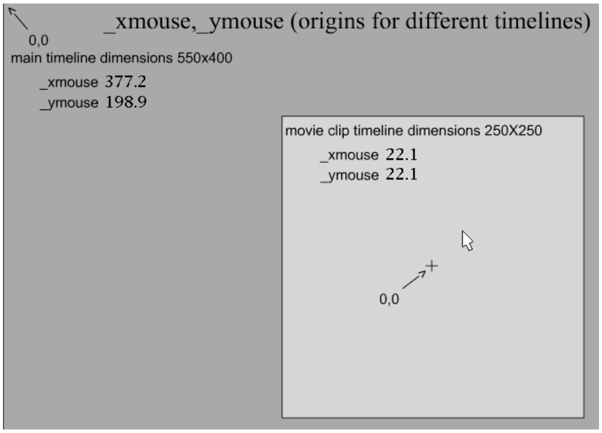
7 Choose Control > Test Movie to use the custom cursor.

Buttons will still function when you use a custom cursor. It’s a good idea to put the custom cursor on the top layer of the Timeline so that it moves in front of buttons and other objects as you move the mouse in the movie.

For more information about the methods of the Mouse object, see their entries in Chapter 7, “ActionScript Dictionary.”

Getting the mouse position

You can use the `_xmouse` and `_ymouse` properties to find the location of the mouse pointer (cursor) in a movie. Each Timeline has an `_xmouse` and `_ymouse` property that returns the location of the mouse within its coordinate system.



The `_xmouse` and `_ymouse` properties within the main Timeline and a movie clip Timeline

The following statement could be placed on any Timeline in the `_level0` movie to return the `_xmouse` position within the main Timeline:

```
x_pos = _root._xmouse;
```

To determine the mouse position within a movie clip, you can use the movie clip's instance name. For example, the following statement could be placed on any Timeline in the `_level0` movie to return the `_ymouse` position in the `myMovieClip` instance:

```
y_pos = _root.myMovieClip._ymouse
```

You can also determine the mouse position within a movie clip by using the `_xmouse` and `_ymouse` properties in a clip action, as in the following:

```
onClipEvent(enterFrame){
   xmousePosition = _xmouse;
   ymousePosition = _ymouse;
}
```

The variables x_pos and y_pos are used as containers to hold the values of the mouse positions. You could use these variables in any script in your movie. In the following example, the values of x_pos and y_pos update every time the user moves the mouse.

```
onClipEvent(mouseMove){
   x_pos = _root._xmouse;
   y_pos = _root._ymouse;
}
```

For more information about the _xmouse and _ymouse properties, see their entries in Chapter 7, "ActionScript Dictionary."

Capturing keypresses

You can use the methods of the predefined Key object to detect the last key the user pressed. The Key object does not require a constructor function; to use its methods, you simply call the object itself, as in the following example:

```
Key.getCode();
```

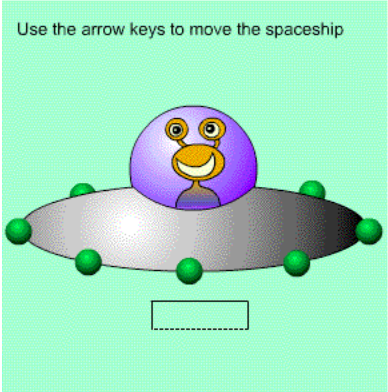
You can obtain either virtual key codes or ASCII values of keypresses:

- 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 can ensure that your movie's controls are the same on every keyboard regardless of language or platform.

ASCII (American Standard Code for Information Interchange) 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.

A common place for using `Key.getCode` is in an `onClipEvent` handler. By passing `keyDown` as the parameter, the handler instructs ActionScript to check for the value of the last key pressed only when a key is actually pressed. This example uses `Key.getCode` in an `if` statement to create navigation controls for the spaceship.



To create keyboard controls for a movie:

1 Decide which keys to use and determine their virtual key codes by using one of these approaches:

- See the list of key codes in Appendix B, "Keyboard Keys and Key Code Values."
- Use a Key object constant. (In the Toolbox list, select Objects, then select Key. Constants are listed in all capital letters.)
- Assign the following clip action, then choose Control > Test Movie and press the desired key:

```
onClipEvent(keyDown) {
    trace(Key.getCode());
}
```

2 Select a movie clip on the Stage.

3 Choose Window > Actions.

4 Double-click the `onClipEvent` action in the Actions category of the toolbox.

5 Choose the `Key down` event in the parameters pane.

6 Double-click the `if` action in the Actions category of the toolbox.

7 Click in the Condition parameter, select Objects; then select Key and `getCode`.

8 Double-click the equality operator (==) in the Operators category of the toolbox.

9 Enter the virtual key code to the right of the equality operator.

Your code should look like this:

```
onClipEvent(keyDown) {
    if (Key.getCode() == 32) {
    }
}
```

10 Select an action to perform if the correct key is pressed.

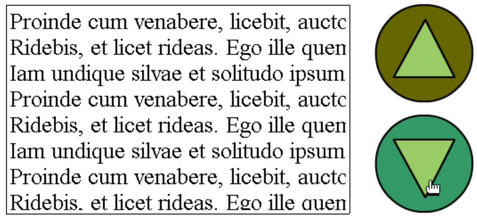
For example, the following action causes the main Timeline to go to the next frame when the Spacebar (32) is pressed:

```
onClipEvent(keyDown) {
    if (Key.getCode() == 32) {
       nextFrame();
    }
}
```

For more information about the methods of the Key object, see their entries in Chapter 7, "ActionScript Dictionary."

Creating a scrolling text field

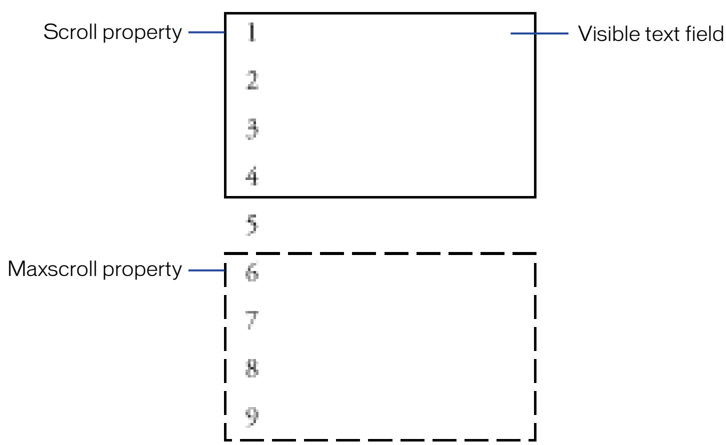
You can use the `scroll` and `maxscroll` properties to create a scrolling text field.



In the Text Options panel, you can assign a variable to any text field set to Input Text or Dynamic Text. The text field acts like a window that displays the value of that variable.

Each variable associated with a text field has a `scroll` and a `maxscroll` property. You can use these properties to scroll text in a text field. The `scroll` property returns the number of the topmost visible line in a text field; you can set and retrieve it. The `maxscroll` property returns the topmost visible line in a text field when the bottom line of text is visible; you can read, but not set, this property.

For example, suppose you have a text field that is four lines long. If it contains the variable `speech`, that would fill nine lines of the text field, and only part of the `speech` variable can be displayed at one time (identified by the solid box):



You can access these properties using dot syntax, as in the following:

```
textFieldVariable.scroll
myMovieClip.textFieldVariable.scroll
textFieldVariable.maxscroll
myMovieClip.textFieldVariable.maxscroll
```

To create a scrolling text field:

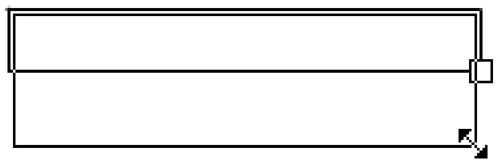
1 Drag a text field on the Stage.

2 Choose Window > Panels > Text Options.

3 Choose Input Text from the pop-up menu.

4 Enter the variable name **text** in the Variable field.

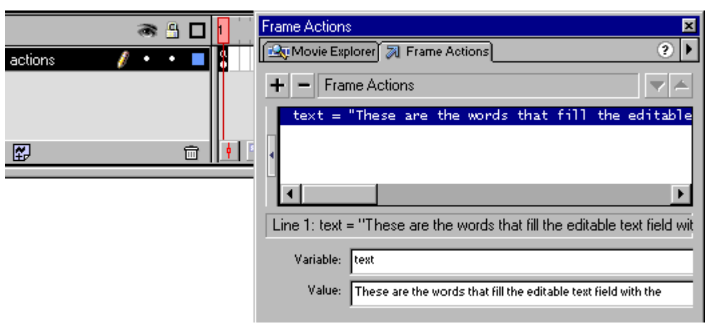
5 Drag the text field's bottom right corner to resize the text field.



6 Choose Window > Actions.

7 Select frame 1 in the main Timeline and assign a `set variable` action that sets the value of `text`.

No text will appear in the field until the variable is set. Therefore, although you can assign this action to any frame, button, or movie clip, it's a good idea to assign the action to frame 1 on the main Timeline, as shown here:



8 Choose Window > Common Libraries > Buttons, and drag a button to the Stage.

9 Press Alt (Windows) or Option (Macintosh) and drag the button to create a copy.

10 Select the top button and choose Window > Actions.

11 Drag the `set variables` action from the toolbox to the Script window in the Actions panel.

12 Enter **text.scroll** in the Variable box.

13 Enter **text.scroll -1** in the Value box and select the Expression check box.

14 Select the Down Arrow button and assign the following `set variables` action:

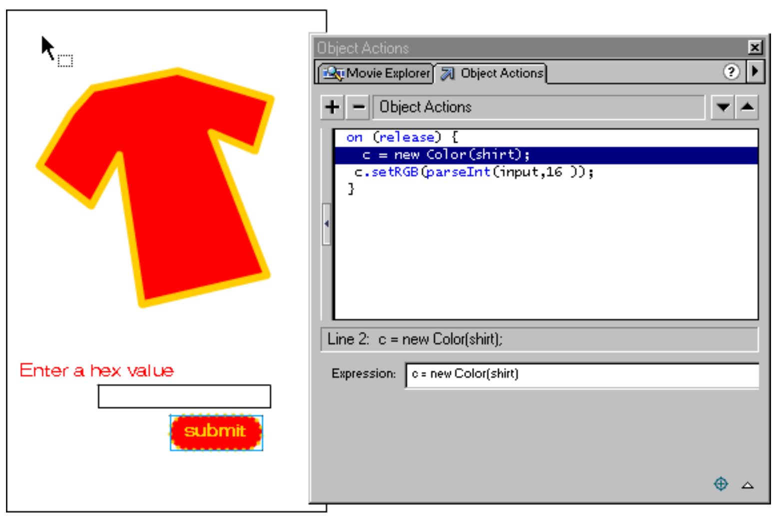
```
text.scroll = text.scroll+1;
```

15 Choose Control > Test Movie to test the scrolling text field.

For more information about the `scroll` and `maxscroll` properties, see their entries in Chapter 7, "ActionScript Dictionary."

Setting color values

You can use the methods of the predefined Color object to adjust the color of a movie clip. The `setRGB` method assigns hexadecimal RGB (red, green, blue) values to the object, and the `setTransform` method sets the percentage and offset values for the red, green, blue, and transparency (alpha) components of a color. The following example uses `setRGB` to change an object's color based on user input.



The button action creates a color object and changes the color of the shirt based on user input

To use the Color object, you need to create an instance of the object and apply it to a movie clip.

To set the color value of a movie clip:

1. Select a movie clip on the Stage, and choose Window > Panels > Instance.
2. Enter the instance name **colorTarget** in the Name box.
3. Drag a text field on the Stage.
4. Choose Window > Panels > Text Options and assign it the variable name **input**.
5. Drag a button to the Stage and select it.
6. Choose Window > Actions.
7. Drag the `set variable` action from the toolbox to the Script window.
8. In the Variable box, enter **c**.
9. In the toolbox, select Objects, then Color, and drag `new Color` to the Value box.
10. Select the Expression check box.
11. Click the Target Path button and select `colorTarget`. Click OK.

    The code in the Script window should look like this:

    ```
    on(release) {
        c = new Color(colorTarget);
    }
    ```

12. Drag the `evaluate` action from the toolbox to the Script window.
13. Enter **c** in the Expression box.
14. In the Objects category of the Toolbox list, select Color; then drag `setRGB` to the Expression box.
15. Select Functions and drag `parseInt` to the Expression box.

    The code should look like this:

    ```
    on(release) {
        c = new Color(colorTarget);
        c.setRGB(parseInt(string, radix));
    }
    ```

16. For the `parseInt` string argument, enter **input**.

    The string to be parsed is the value entered into the editable text field.

17 For the `parseInt` radix argument, enter **16**.

The radix is the base of the number system to be parsed. In this case, 16 is the base of the hexadecimal system that the Color object uses. The code should look like this:

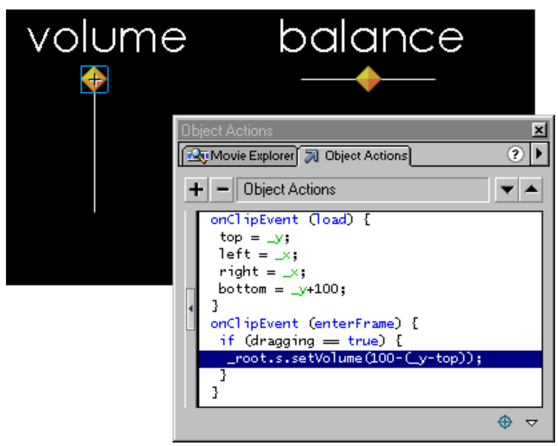
```
on(release) {
    c = new Color(colorTarget);
    c.setRGB(parseInt(input, 16));
}
```

18 Choose Control > Test Movie to change the color of the movie clip.

For more information about the methods of the Color object, see their entries in Chapter 7, "ActionScript Dictionary."

Creating sound controls

To control sounds in a movie, you use the predefined Sound object. To use the methods of the Sound object, you must first create a new Sound object. Then you can use the `attachSound` method to insert a sound from the library into a movie while the movie is running. The Sound object'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.*

To attach a sound to a Timeline:

1 Choose File > Import to import a sound.

2 Select the sound in the library and choose Linkage from the Options menu.

3 Select Export This Symbol and give it the identifier **mySound**.

4 Select frame 1 in the main Timeline and choose Window > Actions.

5 Drag the `set variable` action from the toolbox to the Script window.

6 Enter **s** in the Value box.

7 In the Toolbox list, select Objects, then select Sound, and drag `new Sound` to the Value box.

   The code should look like this:

```
s = new Sound();
```

8 Double-click the `evaluate` action in the toolbox.

9 Enter **s** in the Expression box.

10 In the Objects category of the Toolbox list, select Sound, then drag `attachSound` to the Expression box.

11 Enter "**mySound**" in the ID argument of `attachSound`.

12 Double-click the `evaluate` action in the toolbox.

13 Enter **s** in the Expression box.

14 In the Objects category, select Sound, then drag `start` to the Expression box.

   The code should look like this:

```
s = new Sound();
s.attachSound("mySound");
s.start();
```

15 Choose Control > Test Movie to hear the sound.

To create a sliding volume control:

1 Drag a button to the Stage.

2 Select the button and choose Insert > Convert to Symbol. Choose the movie clip behavior.

   This creates a movie clip with the button on it's first frame.

3 Select the movie clip and choose Edit > Edit Symbol.

4 Select the button and choose Window > Actions.

5 Enter the following actions:

```
on (press) {
    startDrag ("", false, left, top, right, bottom);
    dragging = true;
}
on (release, releaseOutside) {
    stopDrag ();
    dragging = false;
}
```

   The `startDrag` parameters `left`, `top`, `right`, and `bottom` are variables set in a clip action.

6 Choose Edit > Edit Movie to return to the main Timeline.

7 Select the movie clip on the Stage.

8 Enter the following actions:

```
onClipEvent (load) {
    top=_y;
    left=_x;
    right=_x;
    bottom=_y+100;
}

onClipEvent(enterFrame){
    if (dragging==true){
       _root.s.setVolume(100-(_y-top));
    }
}
```

9 Choose Control > Test Movie to use the volume slider.

To create a balance sliding control:

1 Drag a button to the Stage.

2 Select the button and choose Insert > Convert to Symbol. Choose the movie clip property.

3 Select the movie clip and choose Edit > Edit Symbol.

4 Select the button and choose Window > Actions.

5 Enter the following actions:

```
on (press) {
    startDrag ("", false, left, top, right, bottom);
    dragging = true;
}
on (release, releaseOutside) {
    stopDrag ();
    dragging = false;
}
```

The `startDrag` parameters `left`, `top`, `right`, and `bottom` are variables set in a clip action.

6 Choose Edit > Edit Movie to return to the main Timeline.

7 Select the movie clip on the Stage.

8 Enter the following actions:

```
onClipEvent(load){
    top=_y;
    bottom=_y;
    left=_x-50;
    right=_x+50;
    center=_x;
}

onClipEvent(enterFrame){
    if (dragging==true){
       _root.s.setPan((_x-center)*2);
    }
}
```

9 Choose Control > Test Movie to use the balance slider.

For more information about the methods of the Sound object, see their entries in Chapter 7, "ActionScript Dictionary."

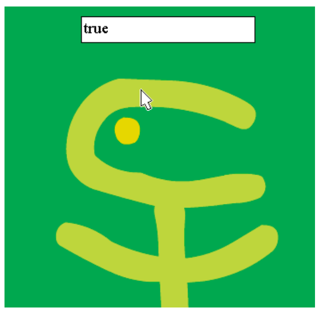
Detecting collisions

You can use the `hitTest` method of the MovieClip object to detect collisions in a movie. The `hitTest` method checks to see if an object has collided with a movie clip and returns a Boolean value (`true` or `false`). You can use the parameters of the `hitTest` method 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.

Each movie clip in a movie is an instance of the MovieClip object. This allows you to call methods of the object from any instance, as in the following:

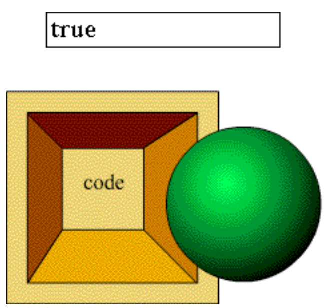
```
myMovieClip.hitTest(target);
```

You can use the `hitTest` method to test the collision of a movie clip and a single point.



"True" appears in the text filed whenever the mouse pointer is over the colored area.

You can also use the `hitTest` method to test a collision between two movie clips.



"True" appears in the text field whenever one movie clip touches the other.

To perform collision detection between a movie clip and a point on the Stage:

1. Select a movie clip on the Stage.
2. Choose Window > Actions to open the Object Actions panel.
3. Double-click `trace` in the Actions category in the toolbox.
4. Select the Expression check box and enter the following in the Expression box:

   ```
   trace (this.hitTest(_root._xmouse, _root._ymouse, true);
   ```

   This example uses the `_xmouse` and `_ymouse` properties as the *x* and *y* coordinates for the hit area and sends the results to the Output window in Test-Movie Mode. You can also set a text field on the Stage to display the results or use the results in an `if` statement.
5. Choose Control > Test Movie and move the mouse over the movie clip to test the collision.

To perform collision detection on two movie clips:

1. Drag two movie clips to the Stage and give them the instance names **mcHitArea** and **mcDrag**.
2. Create a text field on the Stage and enter **status** in the Text Options Variable box.
3. Select `mcHitArea` and choose Window > Actions.
4. Double-click `evaluate` in the toolbox.
5. Enter the following code in the Expression box by selecting items from the toolbox:

   ```
   _root.status=this.hitTest(_root.mcDrag);
   ```
6. Select the `onClipEvent` action in the Script window and choose `enterFrame` as the event.
7. Select `mcDrag` and choose Window > Actions.
8. Double-click `startDrag` in the toolbox.
9. Select the Lock Mouse to Center check box.
10. Select the `onClipEvent` action in the Script window and choose the `Mouse down` event.
11. Double-click `stopDrag` in the toolbox.
12. Select the `onClipEvent` action in the Script window and choose the `Mouse up` event.
13. Choose Control > Test Movie and drag the movie clip to test the collision detection.

For more information about the `hitTest` method, see its entry in Chapter 7, “ActionScript Dictionary.”

4

CHAPTER 4
Working with Movie Clips

A movie clip is a mini Flash movie: it has its own Timeline and properties. A movie clip symbol in the Library may be used multiple times in a Flash movie; each use is called an *instance* of the movie clip. Movie clips can be nested inside each other. To distinguish instances from each other, you can assign each instance an instance name.

Any object can be placed on the Timeline of a movie clip, including other movie clips. Movies that are loaded into the Flash Player using `loadMovie` are also mini Flash movies. Each movie clip, loaded movie, and the main Timeline in a Flash movie are objects with properties and methods that can be manipulated by ActionScript to create complex, nonlinear animation and powerful interactivity.

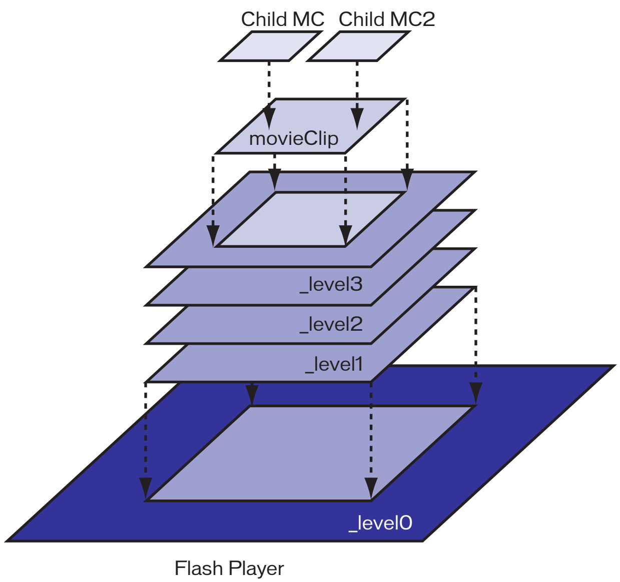
You control movie clips using actions and MovieClip object methods. Actions and methods can be attached to frames or buttons in a movie clip (frame and button actions), or to a specific movie clip instance (clip actions). Actions in a movie clip can control any Timeline in a movie. To control a Timeline you must address it by using a target path. A target path indicates the location of the Timeline in the movie.

You can also turn a movie clip into a "smart" clip; a movie clip with ActionScript that can be reprogrammed without using the Actions panel. Smart clips make it easy to pass objects with complicated ActionScript logic from a programmer to a designer.

About multiple Timelines

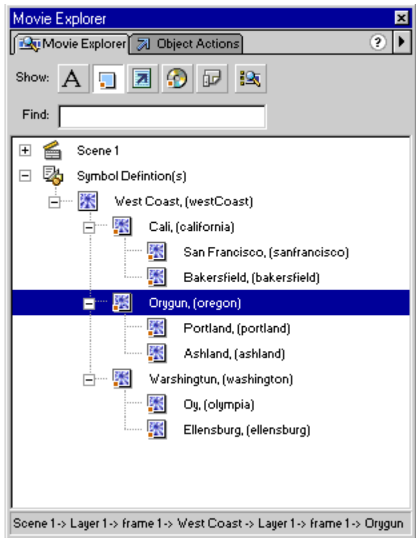
Every Flash movie has a main Timeline located at level 0 in the Flash Player. You can use the `loadMovie` action to load other Flash movies (SWF files) into the Flash Player at any level above level 0 (for example, level 1, level 2, level 15). Each movie loaded into a level of the Flash Player has a Timeline.

Flash movies at any level can have movie clip instances on their Timelines. Each movie clip instance also has a Timeline and can contain other movie clips that also have Timelines. The Timelines of movie clips and levels in the Flash Player are organized hierarchically so that you can organize and easily control the objects in your movie.



The hierarchy of levels and movie clips in the Flash Player

In Flash, this hierarchy of levels and movie clips is called the *display list*. You can view the display list in the Movie Explorer when you are authoring in Flash. You can view the display list in the Debugger when you are playing the movie in Test-Movie Mode, the stand-alone Flash Player, or in a Web browser.



The Movie Explorer shows the hierarchy of Timelines called the "display list"

Timelines in a Flash movie are objects and all have characteristics (properties) and the abilities (methods) of the predefined MovieClip object. Timelines have specific relationships with each other depending on their locations in the display list. Timelines that are nested inside other Timelines are affected by changes made to the Timeline on which they live. For example, if `portland` is a child of `oregon` and you change the `_xscale` property of `oregon`, `portland` will also scale.

Timelines can also send messages to each other. For example, an action on the last frame of one movie clip could tell another movie clip to play.

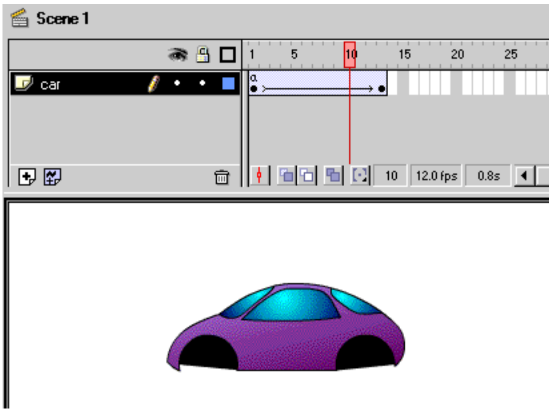
About the hierarchical relationship of Timelines

When you place a `movie clip` instance on another movie clip's Timeline, one movie clip symbol contains the instance of the other movie clip—the first movie clip is the *child* and the second movie clip is its *parent*. The main Timeline of a Flash movie is the parent of all the movie clips on its level.

The parent-child relationships of movie clips are hierarchical. To understand this hierarchy, consider the hierarchy on a computer: the hard drive has a root directory (or folder) and subdirectories. The root directory is analogous to the main Timeline of a Flash movie: it is the parent of everything else. The subdirectories are analogous to movie clips. You can use subdirectories to organize related content.

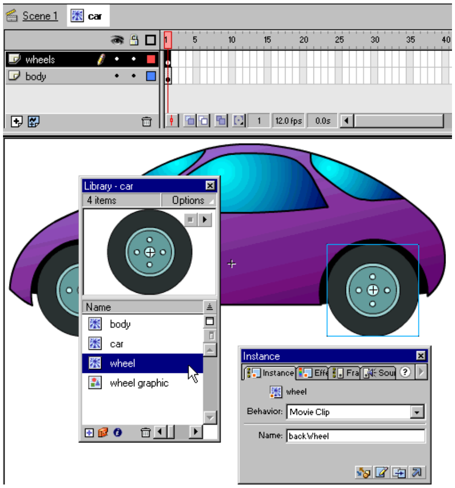
Similarly, you can use the movie clip hierarchy in Flash to organize related visual objects, often in ways similar to the real-world behavior of objects. Any change you make to a parent movie clip is also performed on its children.

For example, you could create a Flash movie that has a car that moves across the Stage. You could use a movie clip symbol to represent the car and set up a motion tween to move it across the Stage.



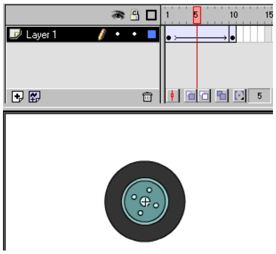
A motion tween moves the car movie clip on the main Timeline

The car is viewed from the side, with two wheels visible. Once you have the car moving, you want to add wheels that rotate. So you make a movie clip for a car wheel, and create two instances of this movie clip, named `frontWheel` and `backWheel`. Then you place the wheels on the Timeline of the car movie clip—not on the main Timeline. As children of `car`, `frontWheel` and `backWheel` are affected by any changes made to `car`. This means that they will move with the car as it tweens across the Stage.



The `frontWheel` and `backWheel` instances are placed on the Timeline of the `car` movie clip.

To make the wheels rotate, you can set up a motion tween to rotate the wheel symbol to make both instances spin. Even after you change `frontWheel` and `backWheel`, they will continue to be affected by the tween on their parent movie clip, `car`; the wheels will spin, but they will also move with the parent movie clip `car` across the Stage.

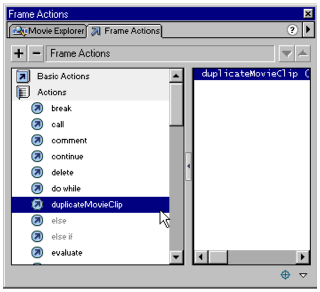


The wheel symbol in Edit Symbol Mode

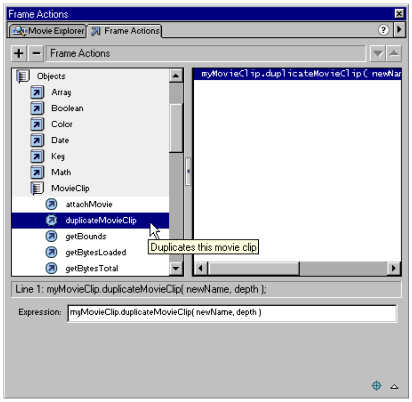
Sending messages between Timelines

You can send messages from one Timeline to another. One Timeline contains the action, called the *controller*, and another receives the action, called the *target*. You can assign an action to a frame or button in a Timeline, or, if the Timeline is a movie clip, to the movie clip itself.

To target Timelines, you can use actions from the Actions category, or you can use methods of the MovieClip object from the Objects category in the Actions panel. For example, you can use the `duplicateMovieClip` action to target and make copies of movie clip instances while a movie plays.



You can use actions from the Actions category to target a Timeline.



You can use methods of the MovieClip object to target a Timeline.

To perform multiple actions on the same target, you can use the `with` action. Similar to the JavaScript `with` statement, the ActionScript `with` action is a wrapper that lets you address the targeted Timeline once, and then have a series of actions execute on that clip; you don't have to address the targeted Timeline in each action.

You can also use the `tellTarget` action to perform multiple actions on the same target.

To communicate between Timelines, you must do the following:

- Enter an instance name for the target movie clip.

  To name a movie clip instance, use the Instance Panel (Window > Panels > Instance). Timelines loaded into levels use their level number as an instance name, for example, `_level6`.

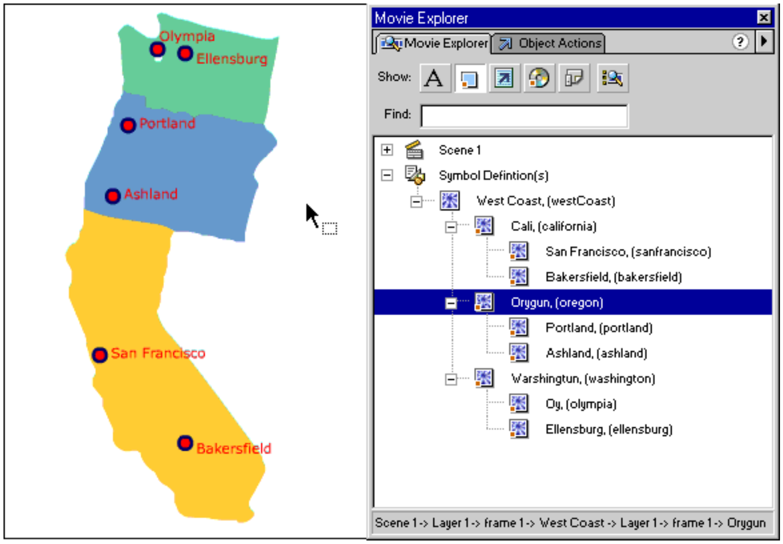
- Enter the target path to the instance name in the Actions Panel.

  You can enter the target path manually, or you can use the Insert Target Path dialog box to target a movie clip. See "Specifying target paths" on page 108.

Note: During playback, a movie clip's Timeline must be on the Stage to be targeted.

About absolute and relative target paths

A target path is the address of the Timeline you want to target. The display list of Timelines in Flash is similar to the hierarchy of files and folders on a Web server.



The Movie Explorer shows the display list of movie clips in Authoring Mode.

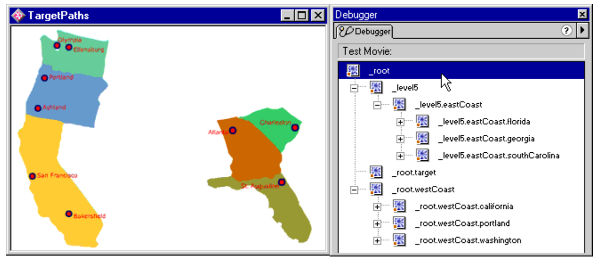
Just as on a Web server, each Timeline in Flash can be addressed two ways: with an absolute path or a relative path. The absolute path of an instance is always the same, regardless of which Timeline calls the action; for example, the absolute path to the instance `california` is always `_level0.westCoast.california`. A relative path is different when called from different locations; for example, the relative path to `california` from `sanfrancisco` is `_parent`, but from `portland`, it's `_parent._parent.california`.

Note: For more information about the Movie Explorer, see *Using Flash*.

An absolute path starts with the name of the level into which the movie is loaded and continues through the display list until it reaches the target instance.

The first movie to be opened in the Flash Player is loaded at level 0. You must assign each additional loaded movie a level number. The target name for a level is `_levelX` where `X` is the level number into which the movie is loaded. For example, the first movie opened in the Flash Player is called `_level0`, a movie loaded into level 3 is called `_level3`.

In the following example, two movies have been loaded into the player, `TargetPaths.swf` at level 0, and `EastCoast.swf` at level 5. The levels are indicated in the Debugger, with level 0 indicated as `_root`.



The Debugger shows the absolute paths of all Timelines in the display list in Test-Movie Mode.

An instance always has the same absolute path, whether it's being called from an action in an instance on the same level, or from an action on a different level. For example, the instance `bakersfield` on level 0 always has the following absolute path in dot syntax:

```
_level0.california.bakersfield
```

In slash syntax, the absolute path substitutes slashes for dots, as in the following:

```
_level0/california/bakersfield
```

To communicate between movies on different levels, you must use the level name in the target path. For example, the `portland` instance would address the `atlanta` instance as follows:

```
_level5.georgia.atlanta
```

In dot syntax, you can use the alias `_root` to refer to the main Timeline of the current level. For the main Timeline, or `_level0`, the _root alias stands for `_level0` when targeted by a clip also on `_level0`. For a movie loaded into `_level5`, `_root` is equal to `_level5` when targeted by a movie clip also on level 1. For example, an action called from the instance `southcarolina` could use the following absolute path to target the instance `florida`:

```
_root.eastCoast.florida
```

In slash syntax, you can use `/` to refer to the main Timeline of the current level, as in the following:

```
/eastCoast/florida
```

In dot syntax in either Absolute or Relative Mode, you can use the same target path rules to identify a variable on a Timeline or a property of an object. For example, the following statement sets the variable name in the instance form to the value "Gilbert":

```
_root.form.name = "Gilbert";
```

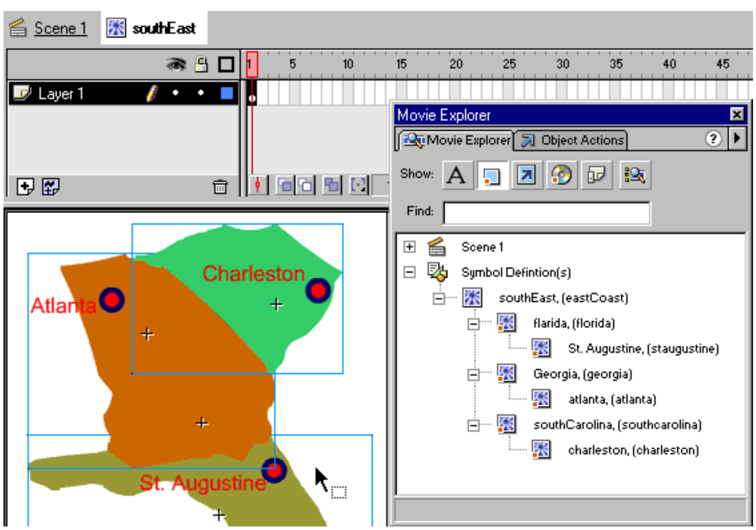
In slash syntax in either Absolute or Relative Mode, you can identify a variable on a Timeline by preceding the variable name with a colon (:), as in the following:

```
/form:name = "Gilbert";
```

A relative path is dependent on the relationship between the controller Timeline and the target Timeline. You can use a relative path to reuse actions because the same action can target different Timelines depending on where the action is placed. Relative paths can address targets only within their own level of the Flash Player; they cannot address movies loaded into other levels. For example, you can't use a relative path in an action on `_level0` that targets a Timeline on `_level5`.

In dot syntax, you can use the keyword `this` in a relative target path to refer to the current Timeline. You can use the alias `_parent` in a relative target path to indicate the parent Timeline of the current Timeline. The `_parent` alias can be used repeatedly to go up one level in the movie clip hierarchy within the same level of the Flash Player. For example, `_parent._parent` controls a movie clip up two levels in the hierarchy.

In the following example, each city (`charleston`, `atlanta`, and `staugustine`) is a child of a state instance and each state (`southcarolina`, `georgia`, and `florida`) is a child of the `eastCoast` instance.



The Movie Explorer shows the parent-child relationships of movie clips.

An action on the Timeline of the instance `charleston` could use the following target path to target the instance `southcarolina`:

```
_parent
```

To target the instance `eastCoast` from an action in `charleston`, you could use the following relative path:

```
_parent._parent
```

In slash syntax, you can use two dots (`..`) to go up a level in the hierarchy. To target `eastCoast` from an action in `charleston`, you could use the following path:

```
../..
```

To target the instance `atlanta` from an action on the Timeline of `charleston`, you could use the following relative path in dot syntax:

```
_parent._parent.georgia.atlanta
```

Relative paths are useful for reusing scripts. For example, you could attach a script to a movie clip that magnifies the movie clip up one level by 150%, as follows:

```
onClipEvent (load) {
   _parent._xscale = 150;
   _parent._yscale = 150;
}
```

You could then reuse this script by placing it on the Timeline of any movie clip.

For more information on addressing and dot syntax, see “Writing Scripts with ActionScript” on page 37.

For more information on Dot syntax and Slash syntax, see “Using ActionScript’s syntax” on page 37.

Specifying target paths

To control a movie clip or loaded movie, you must use a target path to specify a target. A movie clip must have an instance name to be targeted. You can specify a target in several different ways:

- Enter a target path using the Insert Target Path button and dialog box in the Actions panel.
- Enter the target path of the movie clip in your script manually.
- Create an expression by using a reference to a movie clip, or by using the predefined functions `targetPath` and `eval`.

To insert a target path using the Insert Target Path dialog box:

1 Select the movie clip, frame, or button instance to which you want to assign the action.

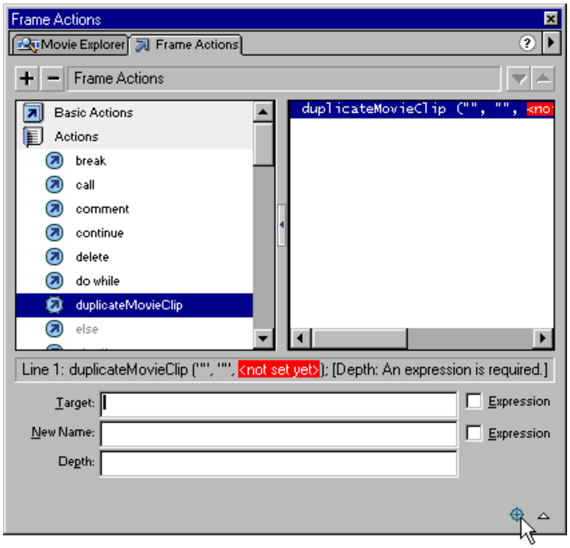
  This will be the controller Timeline.

2 Choose Window > Actions to display the Actions panel.

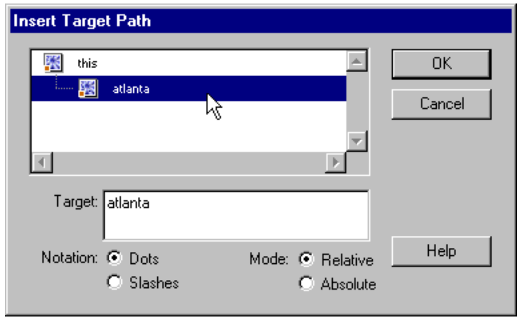
3 In the Toolbox list, choose an action from the Actions category or a method from the MovieClip category inside the Objects category.

4 Click the Target field or location in the script to insert the target path.

5 Click the Insert Target Path button in the bottom right corner of the Actions panel to display the Insert Target Path dialog box.



6 In the Insert Target Path dialog box, choose a syntax: Dots (the default) or Slashes.



7 Choose Absolute or Relative for the target path mode.

See "About absolute and relative target paths" on page 104.

8 Specify your target by doing one of the following:

- Select a movie clip in the Insert Target Path display list.
- Enter a target manually in the Target field using an absolute or relative path and dot syntax.

9 Click OK.

To insert a target path manually:

Follow steps 1-4 above and enter an absolute or relative target path into the Actions panel.

To use an expression as a target path:

1 Follow steps 1-4 above.

2 Do one of the following:

- Manually enter a reference as a target path. A reference is evaluated to determine the target path. You can use a reference as a parameter for the `with` action. In the following example, the variable `index` is evaluated and multiplied by 2. The resulting value is used as the name of the movie clip inside the `Block` instance that is told to play:

  ```
  with (Board.Block[index*2]) {
      play();
  }
  ```

- In the Functions category of the Toolbox list, choose the `targetPath` function.

  The `targetPath` function converts a reference to a movie clip into a string that can be used by actions such as `tellTarget`.

  In the following example, the `targetPath` function converts the reference `Board.Block[index*2+1]` to a string:

  ```
  tellTarget (targetPath (Board.Block[index*2+1])) {
      play();
  }
  ```

  The previous example is equivalent to the following Slash syntax:

  ```
  tellTarget ("Board/Block:" + index*2+1)) {
      play();
  }
  ```

- In the Functions category of the Toolbox list, choose the `eval` function.

  The `eval` function converts a string into a reference to a movie clip that can be used as a target path by actions such as `with`.

  The following script evaluates the variable `i`, adds it to the string `"cat"` and assigns the resulting value to the variable `x` . The variable x is now a reference to a movie clip instance and can call the MovieClip object methods, as in the following:

  ```
  x = eval ("cat" + i);
  x.play();
  ```

  You can also use the `eval` function to call methods directly, as in the following:

  ```
  eval ("cat" + i).play();.
  ```

Using actions and methods to control Timelines

You can use certain actions and methods of the MovieClip object to *target*, or perform tasks on, a movie clip or loaded level. For example, the `setProperty` action sets a property (such as `_width`) of a Timeline to a value (such as `100`). Some MovieClip object methods duplicate the function of all the actions that target Timelines. There are also additional methods, such as `hitTest`, and `swapDepths`. Whether you use an action or a method, the target Timeline must be loaded in the Flash Player when the action or method is called.

The following actions can target movie clips: `loadMovie`, `unloadMovie`, `setProperty`, `startDrag`, `duplicateMovieClip`, and `removeMovieClip`. To use these actions, you must enter a target path in the action's Target parameter to indicate the recipient of the action. Some of these actions can target movie clips or levels and others can only target movie clips.

The following MovieClip object methods can control movie clips or loaded levels and do not have equivalent actions: `attachMovie`, `getBounds`, `getBytesLoaded`, `getBytesTotal`, `globalToLocal`, `localToGlobal`, `hitTest`, and `swapDepths`.

When an action and a method offer similar functions, you can choose to control movie clips by using either one. The choice depends on your preference and familiarity with writing scripts in ActionScript.

For more information about the methods of the MovieClip object and for information about each action, see Chapter 7, "ActionScript Dictionary" on page 157.

About methods versus actions

To use a methods, you invoke it by using the target path to the instance name, followed by a dot, and then the method name and arguments, as in the following statements:

```
myMovieClip.play();
parentClip.childClip.gotoAndPlay(3);
```

In the first statement, the `play` method causes the `myMovieClip` instance to play. In the second statement, the `gotoAndPlay` method sends the playhead in `childClip` (which is a child of the instance `parentClip`) to frame 3 and plays.

Actions that control a Timeline have a Target parameter that specifies the target path. For example, in the following script the `startDrag` action targets the `customCursor` instance and makes it draggable:

```
on(press){
   startDrag("customCursor");
}
```

When you use a method, you call the method at the end of the target path. For example, the following statement performs the same `startDrag` function:

```
customCursor.startDrag();
```

Statements written using the MovieClip object methods tend to be more brief because they don't require the `tellTarget` action. Use of the `tellTarget` action is discouraged because it is not compatible with the ECMA-262 standard.

For example, to tell movie clip `myMovieClip` to start playing using the MovieClip object methods, you would use the following code:

```
myMovieClip.play();
```

The following code produces the same results by using the `tellTarget` action:

```
tellTarget ("myMovieClip") {
   play();
}
```

Using multiple methods or actions to target a Timeline

You can use the `with` action to address a targeted movie clip once, and then execute a series of actions on that clip. The `with` action works on all ActionScript objects (for example Array, Color, and Sound), not just movie clips. The `tellTarget` action is similar to the `with` action. However, the `tellTarget` action is not preferred because it does not work with all ActionScript objects and is not ECMA-262 compliant.

The `with` action takes an object as a parameter. The object that you specify is added to the end of the current target path. All actions nested inside a `with` action are carried out inside the new target path, or *scope*. For example, in the following script on the main Timeline, the `with` action is passed the object `donut.hole` to change the properties of `hole`:

```
with (donut.hole){
   _alpha = 20;
   _xscale = 150;
   _yscale = 150;
}
```

It is as if the statements inside the with action were called from the Timeline of the `hole` instance.

In the following example, note the economy of using the `with` action and the methods of the MovieClip object to issue several instructions:

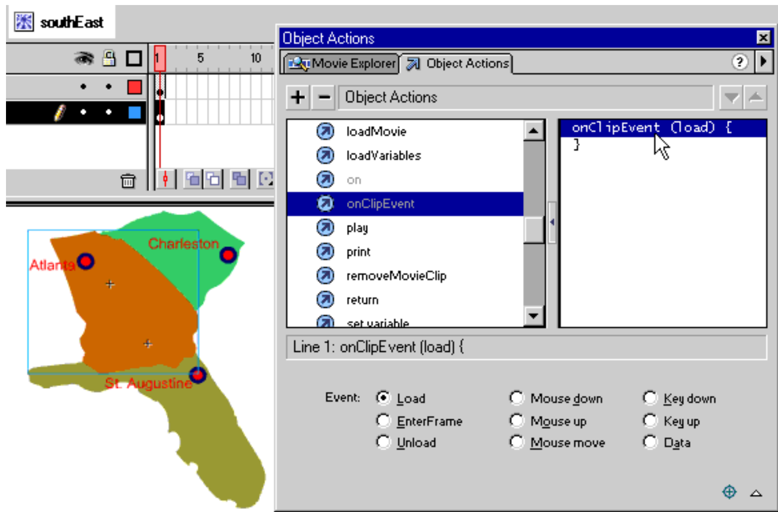
```
with (myMovieClip) {
   _x -= 10;
   _y += 10;
   gotoAndPlay(3);
}
```

For more information on the `tellTarget` action, see *Using Flash.*

Assigning an action or method

Actions and methods can be assigned to a button or frame in a Timeline, or to a movie clip instance.

To assign an action or method to a movie clip instance you must use an `onClipEvent` handler. All actions attached to the instance are nested inside an `onClipEvent` handler and execute after it is triggered. The `onClipEvent` action is triggered by either Timeline events (such as loading a movie) or user events (such as a mouse click or keypress). For example, `onClipEvent(mouseMove)` triggers an action every time the user moves the mouse.



The `onClipEvent` *action is assigned to an instance on the Stage. The* `onClipEvent` *events are listed in the Parameters pane in the Actions panel.*

Loading and unloading additional movies

You can use the `loadMovie` action or method to play additional movies without closing the Flash Player, or to switch movies without loading another HTML page. You can also use `loadMovie` to send variables to a CGI script, which generates a SWF file as its CGI output. When you load a movie, you can specify a level or movie clip target into which the movie will load.

The `unloadMovie` action and method removes a movie previously loaded by `loadMovie`. Explicitly unloading movies with `unloadMovie` ensures a smooth transition between movies and can lighten the memory required by the Flash Player. Use the `loadMovie` action to do any of the following:

- Play a sequence of banner ads that are SWF files by placing a `loadMovie` action at the end of each SWF file to load the next movie
- Develope a branching interface that lets the user choose among several different SWF files
- Build a navigation interface with navigation controls in level 0 that load other levels. Loading levels produces smoother transitions than loading new HTML pages in a browser.

Changing movie clip position and appearance

To change the properties of a movie clip as it plays, you can use the `setProperty` action or write a statement that assigns a value to a property. If you load a movie into a target, the loaded movie inherits the properties of the targeted movie clip. Once the movie is loaded, you can change those properties.

Some properties, called *read-only* properties, have values that you can read but not set. 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 main Timeline and sets the `_x` property of the `customCursor` instance to that value:

```
onClipEvent(enterFrame){
   customCursor._x = _root._xmouse;
}
```

You can also use the `getProperty` function to retrieve movie clip properties.

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 only belong to the level 0 Timeline. All other properties belong to each movie clip or loaded level. The table below lists all the movie clip properties:

Properties			
_alpha	_highquality	_totalframes	_xscale
_currentframe	_name	_url	_y
_droptarget	_quality	_visible	_ymouse
_focusrect	_rotation	_width	_yscale
_framesloaded	_soundbuftime	_x	
_height	_target	_xmouse	

Dragging movie clips

You can use the `startDrag` action or method to make a movie clip draggable while a movie is playing. 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 can be dragged at a time.

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 was dragged to a specific movie clip (such as a "trash can" movie clip), and then trigger another action. See "Using "if" statements" on page 60 and "Using operators to manipulate values in expressions" on page 51.

Duplicating and removing movie clips

You can create or remove movie clip instances as your movie is playing using `duplicateMovieClip` or `removeMovieClip`, respectively. The `duplicateMovieClip` action and method dynamically create a new instance of the movie clip, assigning it a new instance name and giving it a depth. A duplicated movie clip always starts at frame 1 even if the original movie clip was on another frame when duplicated, and is always on top of all predefined movie clips placed on the Timeline. Variables are not copied into the duplicate movie clip.

To delete a movie clip you created with `duplicateMovieClip`, use `removeMovieClip`. Duplicated movie clips also are removed if the parent movie clip is deleted.

Attaching movie clips

You can retrieve a copy of a movie clip from a library and play it as part of your movie using the `attachMovie` method. This method loads another movie clip into your movie clip and plays it as the movie runs.

To use the `attachMovie` method, the movie clip being attached must be given a unique name in the Symbol Linkage Properties dialog box.

To name a movie clip for sharing:

1. Select the movie clip in the movie's Library that you want to attach.
2. In the Library window, choose Linkage from the Options menu.
3. For Linkage, choose Export This Symbol.
4. In the Symbol Linkage Properties dialog box, for Identifier, enter a name for the movie clip. The name must differ from the symbol's name in the library.
5. Click OK.

To attach a movie clip to another movie clip:

1. In the Actions panel, specify the target to which you want to attach a movie clip.
2. In the Toolbox list, select the MovieClip object and then select the `attachMovie` method.
3. Set the following arguments:

- For `idName`, specify the Identifier name that you entered in the Symbol Linkage Properties dialog box.
- For `newName`, enter an instance name for the attached clip so that you will be able to target it.
- For `depth`, enter the level at which the duplicate movie will be attached to the movie clip. Each attached movie has its own stacking order, with level 0 as the level of the originating movie. Attached movie clips are always on top of the original movie clip.

  For example:

  ```
  myMovieClip.attachMovie("calif", "california", 10 );
  ```

Creating smart clips

A "smart" clip is a movie clip with defined clip parameters that can be changed. Those parameters are then passed to actions in the smart clip that change the clip's behavior.

To create a smart clip, you assign clip parameters to a movie clip symbol in the Library. You can write ActionScript statements in the smart clip that operate on the clip parameters, much like you use arguments in a function definition. You can select a smart clip instance on the Stage and change the values of the parameters in the Clip Parameters panel. During playback, the values set in the panel are sent to the smart clip before any actions in the movie are executed.

Smart clips are useful for passing complicated Flash elements from a programmer to a designer. The programmer can write actions in the smart clip with variables that control the clip and the movie. A designer can then change the values of those variables in the Clip Parameters panel without having to open the Actions panel.

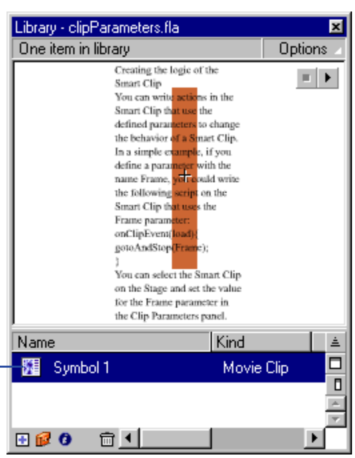
You can use smart clips to create interface elements—such as radio buttons, pop-up menus, tooltips, surveys, games, and avatars. Any movie clip that you want to reuse in a different way without changing the scripts would be a good smart clip.

In addition, you can create a custom interface in Flash for the Clip Parameters panel to facilitate designers who are customizing the clip.

Defining clip parameters

Clip parameters are pieces of data that are passed to a movie clip when it loads in a movie. You can define clip parameters when you are authoring your movie. You can use these parameters in actions to change the appearance and behavior of the smart clip while the movie is playing. A special icon in the Library window indicates a movie clip with defined clip parameters.

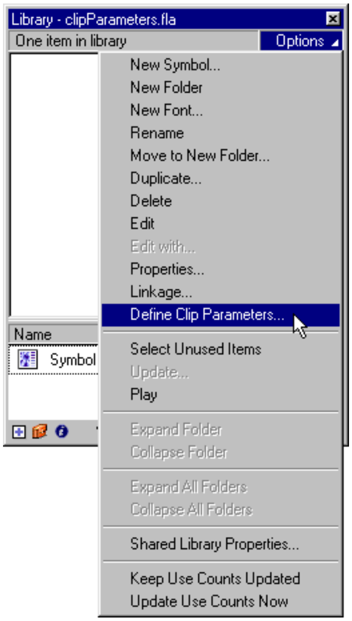
Smart Clip icon



To define clip parameters for a movie clip:

1 Select a movie clip symbol in your movie's library and do one of the following to display the Clip Parameters dialog box:

- Right-click (Windows) or Control-click (Macintosh), and choose Define Clip Parameters from the context menu.
- Choose Define Clip Parameters from the Options menu at the upper right of the Library window.



2 Use the controls in the Clip Parameters dialog box as follows:

- Click the Add (+) button to add a new name/value pair or additional parameters for a selected name/value pair.
- Click the Minus (-) button to delete a name/value pair.
- Use the arrow buttons to change the order of parameters in the list.
- Select a field by double-clicking it, and then enter a value.

3 For Name, enter a unique identifier for the parameter.

4 For Type, choose the kind of data the parameter will contain from the pop-up menu:

- Select Default to use a string or number value.
- Select Array for a dynamic list of items that can grow or shrink.
- Select Object to declare several related elements with names and values, such as a Point object with *x* and *y* elements.
- Select List to limit the selection to several choices, such as `true` or `false` or `Red`, `Green`, or `Blue`.

5 For Value, select the default value that the parameter will contain from the pop-up menu.

6 If you want to use a custom interface for the Clip Parameters panel, do one of the following:

- Enter a relative path to the custom interface SWF file in the Link to Custom UI field.
- Click the Link to Custom UI folder, and browse to the custom interface SWF file.

See “Creating a custom interface” on page 123.

7 For Description, enter notes to that will appear in the Clip Parameters panel that describe what each parameter does.

You can include any information in the Description that you want someone who uses the smart clip to know. For example, an explanation of methods you have defined.

8 Choose Lock in Instance to prevent users from renaming the parameters in the Clip Parameters panel.

It is recommended that you leave the parameter names locked.

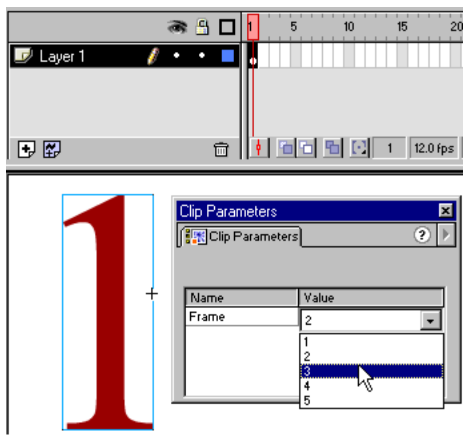
9 Click OK.

Setting clip parameters

You can write actions in the smart clip that use the defined parameters to change the behavior of a smart clip. In a simple example, if you define a clip parameter with the name `Frame`, you could write the following script on the smart clip that uses the `Frame` parameter:

```
onClipEvent(load){
   gotoAndStop(Frame);
}
```

You can then select the Smart Clip on the Stage and set the value for the `Frame` parameter in the Clip Parameters panel to change which frame is played.



To set a smart clip's clip parameters:

1 Select a smart clip instance on the Stage.

   Smart clips are movie clips, so only the first frame will display in authoring mode.

2 Choose Window > Panels > Clip Parameters to display the Clip Parameters panel.

3 In the Clip Parameters panel, do one of the following:

- Double-click the Value field to select it and enter a value for each parameter.

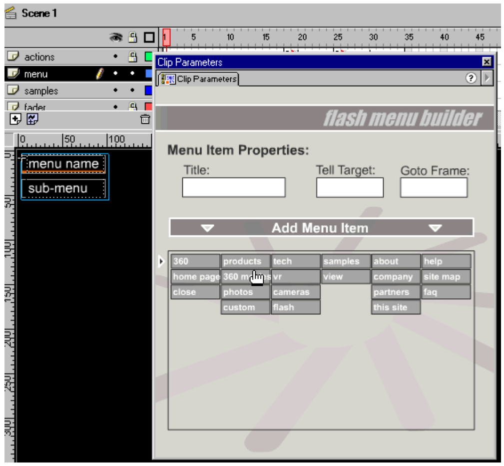
   If the parameter has been defined as a List, a pop-up menu will appear.

- If a custom interface has been defined, use the interface elements provided.

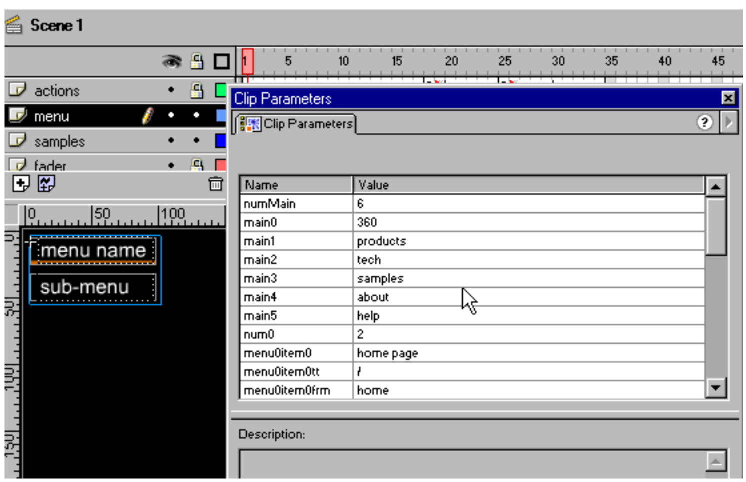
4 Choose Control > Test Movie to see the smart clip's behavior change.

Creating a custom interface

A custom interface is a Flash movie that lets you enter values to be passed to the smart clip. The custom interface replaces the interface of the Clip Parameters panel.



The Clip Parameters panel with a custom interface movie.



The same smart clip without a custom interface in the Clip Parameters panel

Any values you enter using a custom interface are passed from the Clip Parameters panel to the smart clip through an intermediary, or exchange, movie clip in the custom interface. The exchange movie clip must have the instance name `xch`. If a custom interface is selected in the Define Clip Parameters dialog box, the smart clip instance passes the defined parameters to the `xch` movie clip and any new values entered in the custom interface are copied to `xch` and passed back to the smart clip.

You must place the `xch` clip on the main Timeline of the interface movie and `xch` must always be loaded. The `xch` movie clip should contain only the values to be passed to the Smart Clip. It should not contain any graphics, other movie clips, or ActionScript statements; `xch` is merely a container through which values are passed. You can transfer top-level objects, such as Arrays and Objects, through the `xch` clip. However, you should not pass nested Arrays or Objects.

To create a custom interface for a Smart Clip:

1. Choose File > New to create a new Flash movie.
2. Choose Insert > New Symbol to create the exchange movie clip.
3. Create a new layer called “Exchange Clip”.
4. With the “Exchange Clip” layer selected, drag the exchange movie clip from the Library window to the Stage in frame 1.
5. Select the exchange movie clip on the Stage, choose Window > Panels > Instance, and enter the name `xch`.
6. Create the interface elements that the author will interact with to set the clip parameters. For example, a pop-up menu, radio buttons, or drag-and-drop menu items.
7. Use the `set variable` action to copy variable and object values to the `xch` instance.

   For example, if a button is used as an interface element, the button could have an action that sets the value of the variable `vertical` and passes it to `xch`, as in the following:

   ```
   on (release){
       _root.xch.vertical = true;
   }
   ```
8. Export the movie as a SWF file.

To use custom interface SWF with a Smart Clip, you need to link them in the Define Clip Parameters dialog box in the library that contains the Smart Clip. It’s a good idea to save the SWF file in the same directory as the FLA containing the Smart Clip. If you reuse the Smart Clip in another file or pass the Smart Clip to another developer, the Smart Clip and the custom interface SWF must remain in the same relative locations.

5

CHAPTER 5
Integrating Flash with Web Applications

Flash movies can send information to and load information from remote files. To send and load variables, you use the `loadVariables` or `getURL` action. To load a Flash Player movie from a remote location, you use the `loadMovie` action. To send and load XML data, you use the XML or XMLSocket object. You can structure XML data using the predefined XML object methods.

You can also create Flash forms consisting of common interface elements, such as text fields and pop-up menus, to collect data that will be sent to a server-side application.

To extend Flash so that it can send and receive messages from the movie's host environment—for example, the Flash Player or a JavaScript function in a Web browser—you can use `fscommand` and Flash Player methods.

Sending and loading variables to and from a remote file

A Flash movie is a window for capturing and displaying information, much like an HTML page. Flash movies, unlike HTML pages, can stay loaded in the browser and continuously update with new information without having to refresh. You can use Flash actions and object methods to send information to and receive information from server-side scripts, text files, and XML files.

Server-side scripts can request specific information from a database and relay it back and forth between the database and a Flash movie. Server-side scripts can be written in many different languages: some of the most common are Perl, ASP (Microsoft Active Server Pages), and PHP.

Storing information in a database and retrieving it allows you to create dynamic and personalized content for your movie. For example, you could create a message board, personal profiles for users, or a shopping cart that remembers what a user has purchased so that it can determine the user's preferences.

You can use several ActionScript actions and object methods to pass information into and out of a movie. Each action and method uses a protocol to transfer information. Each also requires information to be formatted in a certain way.

The following actions use HTTP or HTTPS protocol to send information in URL encoded format: `getURL`, `loadVariables`, `loadMovie`.

The following methods use HTTP or HTTPS protocol to send information as XML: `XML.send`, `XML.load`, `XML.sendAndLoad`.

The following methods create and use a TCP/IP socket connection to send information as XML: `XMLSocket.connect`, `XMLSocket.send`.

About security

When playing a Flash movie in a Web browser, you can load data into the movie only from a file that is on a server in the same subdomain. This prevents Flash movies from being able to download information from other people's servers.

To determine the subdomain of a URL consisting of one or two components, use the entire domain:

Domain	Subdomain
http://macromedia	macromedia
http://macromedia.com	macromedia.com

To determine the subdomain of a URL consisting of more than two components, remove the last level:

Domain	Subdomain
http://x.y.macromedia.com	y.macromedia.com
http://www.macromedia.com	macromedia.com

The following chart shows how the Flash Player determines whether or not to permit an HTTP request:

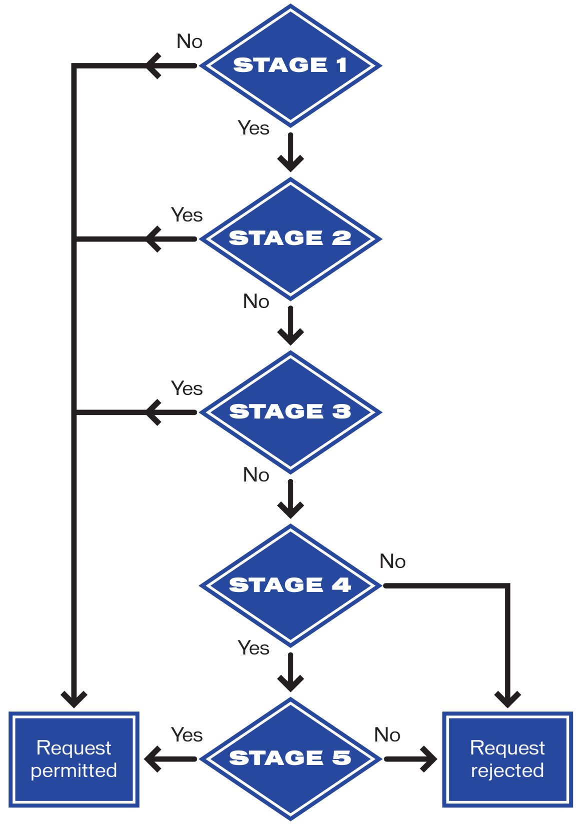
STAGE 1
Is this request for:
loadVariables,
xml.load,
xml.sendAndLoad,
or
xmlsocket.connect?

STAGE 2
Is the request for
a relative URL?

STAGE 3
Is the requesting
movie loaded from
a local disk?
(Its URL begins with
file: or res:)

STAGE 4
Does the URL being
requested start with
http://,
https:// or
ftp://?

STAGE 5
Does the domain name
of the requesting movie
match the domain name
of the requested URL?



When you use the XMLSocket object to create a socket connection with a server, you must use a port numbered 1024 or higher. (Ports with lower numbers are commonly used for Telnet, FTP, the World Wide Web, or Finger.)

Flash relies on standard browser and HTTP and HTTPS security features. Essentially, Flash offers the same security that is available with standard HTML. You should follow the same rules that you follow when building secure HTML Web sites. For example, to support secure passwords in Flash, you need to establish your password authentication with a request to a Web server.

To create a password, use a text field to request a password from the user. Submit it to a sever in a `loadVariables` action or in an `XML.sendAndLoad` method using an HTTPS URL with the `POST` method. The Web server can then verify whether the password is valid. This way, the password will never be available in the SWF file.

Checking for loaded data

Each action and method that loads data into a movie (except `XMLSocket.send`) is *asynchronous*; the results of the action are returned at an indeterminate time.

Before you can use loaded data in a movie, you must check to see if it has been loaded. For example, you can't load variables and manipulate the values of those variables in the same script. In the following script, you can't use the variable `lastFrameVisited` until you're sure the variable has loaded from the file `myData.txt`:

```
loadVariables("myData.txt", 0);
gotoAndPlay(lastFrameVisited);
```

Each action and method has a specific technique you can use to check data it has loaded. If you use the `loadVariables` or `loadMovie` actions you can load information into a movie clip target and use the `data` event of the `onClipEvent` action to execute a script. If you use the `loadVariables` action to load the data, the `onClipEvent(data)` action executes when the last variable is loaded. If you use the `loadMovie` action to load the data, the `onClipEvent(data)` action executes each time a fragment of the movie is streamed into the Flash Player.

For example, the following button action loads the variables from the file `myData.txt` into the movie clip `loadTargetMC`:

```
on(release){
   loadVariables("myData.txt", _root.loadTargetMC);
}
```

An action assigned to the `loadTargetMC` instance uses the variable `lastFrameVisited`, which is loaded from the file `myData.txt`. The following action will execute only after all the variables, including `lastFrameVisited`, are loaded:

```
onClipEvent(data) {
   goToAndPlay(lastFrameVisited);
}
```

If you use the `XML.load` and `XMLSocket.connect` methods, you can define a handler that will process the data when it arrives. A handler is a property of the XML or XMLSocket object to which you assign a function that you have defined. The handlers are called automatically when the information is received. For the XML object, use `XML.onLoad`. For the XMLSocket object, use `XMLSocket.onConnect`.

For more information, see "Using the XML object" on page 132 and "Using the XMLSocket object" on page 135.

Using loadVariables, getURL, and loadMovie

The `loadVariables`, `getURL`, and `loadMovie` actions all communicate with server-side scripts using the HTTP protocol. Each action sends all the variables from the Timeline to which the action is attached; each action handles its response as follows:

- `getURL` returns any information to a browser window, not into the Flash Player.
- `loadVariables` loads variables into a specified Timeline in the Flash Player.
- `loadMovie` loads a movie into a specified level in the Flash Player.

When you use the `loadVariables`, `getURL`, or `loadMovie` actions, you can specify several arguments:

- *`URL`* is the file in which the remote variables reside.
- *`Location`* is the level or target in the movie that receives the variables.

  For more information about levels and targets, see "About multiple Timelines" on page 98.

  Note: The `getURL` action does not take this argument.

- *`Variables`* sets the HTTP method, either `GET` or `POST`, by which the variables will be sent.

For example, if you wanted to track the high scores for a game, you could store the scores on a server and use a `loadVariables` action to load them into the movie each time someone played the game. The action might look something like this:

```
loadVariables("http://www.mySite.com/scripts/high_score.php",
_root.scoreClip, GET);
```

This loads variables from the PHP script called `high_score.php` into the movie clip instance `scoreClip` using the `GET` HTTP method.

Any variables loaded with the `loadVariables` action must be in the standard MIME format application/x-www-urlformencoded (a standard format used by CGI scripts). The file that you specify in the URL argument of the `loadVariables` action must write out the variable and value pairs in this format so that Flash can read them.

The 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, this phrase defines several variables:

```
highScore1=54000&playerName1=rockin+good&highScore2=53455&playerN
ame2=bonehelmet&highScore3=42885&playerName3=soda+pop
```

For more information on `loadVariables`, `getURL`, and `loadMovie`, see their entries in Chapter 7, "ActionScript Dictionary."

About XML

XML (*Extensible Markup Language*) is becoming the standard for the interchange of structured data in Internet applications. You can integrate data in Flash with servers that use XML technology to build sophisticated applications, such as a chat system or a brokerage system.

In XML, as with HTML, you can use tags to *markup*, or specify, a body of text. In HTML, you can 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's displayed. This allows the same XML document to be used and reused in different environments.

Every XML tag is called a *node*, or an element. Each node has a type (1–XML element, or 3-text node) and elements may also have attributes. A node nested in a node is called a *child* or a *childNode*. This hierarchical tree structure of nodes is called the XML DOM (Document Object Model)—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 childNode `<HOLDING>`, which has the attributes `SYMBOL`, `QTY`, `PRICE`, and `VALUE`:

```
<PORTFOLIO>
   <HOLDING SYMBOL="RICH"
      QTY="75"
      PRICE="245.50"
      VALUE="18412.50" />
</PORTFOLIO>
```

Using the XML object

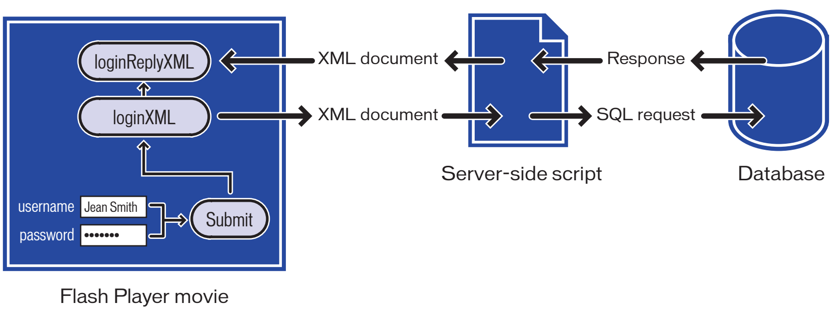
You can use the methods of the ActionScript XML object (for example, `appendChild`, `removeNode`, and `insertBefore`) to structure XML data in Flash to send to a server and to manipulate and interpret downloaded XML data.

You can use the following XML object methods to send and load XML data to a server via the HTTP `POST` method:

- `load` downloads XML from a URL and places it in an ActionScript XML object.
- `send` passes an XML object to a URL. Any returned information is sent to another browser window.
- `sendAndLoad` 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 for trading securities 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 Flash movie (for example, a username 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 movie.



The flow and conversion of data between a Flash Player movie, a server-side scripting document, 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 sends the XML objects attached to the Submit button in the form.

When users enter their information into text fields in the Flash movie 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 presses the Submit button, the `loginXML` object is converted to a string of XML and sent to the server.

The following script is attached to the Submit button. To understand the script, read the commented lines of each script as indicated by the characters `//`:

```
on (release) {
   // A. Construct a XML document with a LOGIN element
   loginXML = new XML();
   loginElement = loginXML.createElement("LOGIN");
   loginElement.attributes.username = username;
   loginElement.attributes.password = password;
   loginXML.appendChild(loginElement);

   // B. Construct a XML object to hold the server's reply
   loginReplyXML = new XML();
   loginReplyXML.onLoad = onLoginReply;

   // C. Send the LOGIN element to the server,
   //    place the reply in loginReplyXML
   loginXML.sendAndLoad("https://www.imexstocks.com/main.cgi",
            loginReplyXML);
}
```

The first section of the script generates the following XML when the user presses the SUBMIT button:

```
<LOGIN USERNAME="JeanSmith" PASSWORD="VerySecret" />
```

The server receives the XML, generates an XML response, and sends it back to the Flash movie. If the password is accepted, the server responds with the following:

```
<LOGINREPLY STATUS="OK" SESSION="rnr6f7vkj2oe14m7jkkycilb" />
```

This XML includes a `SESSION` attribute which 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:

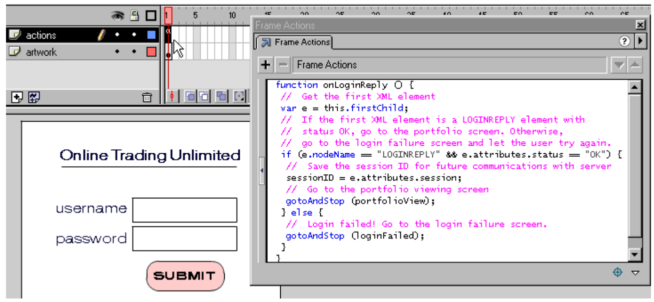
```
<LOGINREPLY STATUS="FAILED" />
```

The `LOGINREPLY` XML node must load into a blank XML object in the Flash movie. The following statement creates the XML object `loginreplyXML` to receive the XML node:

```
// B. Construct an XML object to hold the server's reply
loginReplyXML = new XML();
loginReplyXML.onLoad = onLoginReply;
```

The second statement assigns the `onLoginReply` function to the `loginReplyXML.onLoad` handler.

The `LOGINREPLY` XML element arrives asynchronously, much like the data from a `loadVariables` action, and loads into the `loginReplyXML` object. When the data arrives, the `onLoad` method of the `loginReplyXML` object is called. You must define the `onLoginReply` function and assign it to the `loginReplyXML.onLoad` handler so that it can process the `LOGINREPLY` element. The `onLoginReply` function is assigned to the frame that contains the submit button.



The `onLoginReply` function is defined on the first frame of the movie.

The `onLoginReply` function is defined in the first frame of the movie. To understand the script, read the commented lines of each script as indicated by the characters `//`:

```
function onLoginReply() {
   // Get the first XML element
   var e = this.firstChild;
   // If the first XML element is a LOGINREPLY element with
   // status OK, go to the portfolio screen.  Otherwise,
   // go to the login failure screen and let the user try again.
   if (e.nodeName == "LOGINREPLY" && e.attributes.status == "OK") {
// Save the session ID for future communications with server
   sessionID = e.attributes.session;
// Go to the portfolio viewing screen
      gotoAndStop("portfolioView");
   } else {
      // Login failed!  Go to the login failure screen.
      gotoAndStop("loginFailed");
   }
}
```

The first line of this function, `var e = this.firstChild`, uses the keyword `this` to refer to the XML object `loginReplyXML` that has just been loaded with XML from the server. You can use `this` because `onLoginReply` has been invoked as `loginReplyXML.onLoad`, so even though `onLoginReply` appears to be a plain function, it actually behaves as a method of `loginReplyXML`.

To send the username and password as XML to the server and to load an XML response back into the Flash movie, you can use the `sendAndLoad` method, as in the following:

```
// C. Send the LOGIN element to the server,
//    place the reply in loginReplyXML
   loginXML.sendAndLoad("https://www.imexstocks.com/main.cgi",
loginReplyXML);
```

For more information about XML methods, see their entries in Chapter 7, “ActionScript Dictionary.”

Note: This design is only an example, and we 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.

Using the XMLSocket object

ActionScript provides a predefined XMLSocket object that allows you to open a continuous connection with a server. A socket connection allows the server to 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 in XML format. You can use the XML object 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 Flash movie. This type of server-side application can be written in a programming language such as Java.

You can use the ActionScript XMLSocket object’s `connect` and `send` methods 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 XMLSocket object’s `connect` method, the Flash Player opens a TCP/IP connection to the server and keeps that connection open until one of the following happens:

- The `close` method of the XMLSocket object is called.
- No more references to the XMLSocket object exist.
- The Flash Player quits.
- 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 of each script as indicated by the characters `//`:

```
//create a new XMLSocket object
sock = new XMLSocket();
//call its connect method to establish a connection with port 1024
//of the server at the URL
sock.connect("http://www.myserver.com", 1024);
//define a function to assign to the sock object that handles
//the servers response. If the connection succeeds, send the myXML
//object. If it fails, provide an error message in a text field.
function onSockConnect(success){
   if (success){
      sock.send(myXML);
   } else {
      msg="There has been an error connecting to "+serverName;
   }
}
//assign the onSockConnect function to the onConnect property
sock.onConnect = onSockConnect;
```

For more information, see the entry for XMLSocket in Chapter 7, "ActionScript Dictionary."

Creating forms

Flash forms provide an advanced type of interactivity—a combination of buttons, movies, and text fields that let you pass information to another application on a local or remote server. All common form elements (such as radio buttons, drop-down lists, and check boxes) can be created as movies or buttons with the look and feel of your Web site's overall design. The most common form element is an input text field.

Common types of forms that use such interface elements include chat interfaces, order forms, and search interfaces. For example, a Flash form can collect address information and send it to another application that compiles the information into an e-mail message or database file. Even a single text field is considered a form and can be used to collect user input and display results.

Forms require two main components: the Flash interface elements that make up the form and either a server-side application or client-side script to process the information that the user enters. The following steps outline the general procedure for creating a form in Flash.

To create a form:

1. Place interface elements in the movie using the layout you want.

   You can use interface elements from the Buttons-Advanced common library or create your own.

2. In the Text Options panel, set text fields to Input and assign each a unique variable name.

   For more information about creating editable text fields, see *Using Flash.*

3. Assign an action that either sends, loads, or sends and loads the data.

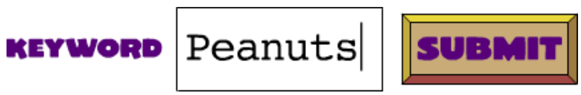
Creating a search form

An example of a simple form is a search field with a Submit button. As an introduction to creating forms, the following example provides instructions for creating a search interface using a `getURL` action. By entering the required information, users can pass a keyword to a search engine on a remote Web server.

To create a simple search form:

1. Create a button for submitting the entered data.
2. Create a label, a blank text field, and an instance of the button on the Stage.

   Your screen should look like this:



3. Select the text field and choose Window > Panels > Text Options.
4. In the Text Options panel, set the following options:

- Choose Input Text from the pop-up menu.
- Select Border/Bg.
- Specify a variable name.

Note: Individual search engines may require a specific variable name. Go to the search engine's Web site for details.

5. On the Stage, select the button and choose Window > Actions.

   The Object Actions panel appears.

Note: A check next to Actions in the Window menu indicates the panel is open.

6. Drag the `getURL` action from the toolbox to the Script window.
7. In the Parameters pane, set the following options:

- For URL, enter the URL of the search engine.
- For Window, select `_blank`. This will open a new window that displays the search results.
- For Variables, select Send Using GET.

8. To test the form, choose File > Publish Preview > HTML.

Using variables in forms

You can use variables in a form to store user input. To set variables, you use editable text fields or assign actions to buttons in interface elements. For example, each item in a pop-up menu is a button with an action that sets a variable to indicate the selected item. You can assign a variable name to an input text field. The text field acts like a window that displays the value of that variable.

When you pass information to and from a server-side script, the variables in the Flash movie must match the variables in the script. For example, if the script expects a variable called `password`, the text field into which users enter the password should be given the variable name `password`.

Some scripts require hidden variables, which are variables that the user never sees. To create a hidden variable in Flash, you can set a variable on a frame in the movie clip that contains the other form elements. Hidden variables are sent to the server-side script along with any other variables set on the Timeline that contains the action that submits the form.

Verifying entered data

For a form that passes variables to an application on a Web server, you'll want to verify that users are entering proper information. For example, you don't want users to enter text in a phone number field. Use a series of `set variable` actions in conjunction with `for` and `if` to evaluate entered data.

The following sample action checks to see whether the entered data is a number, and that the number is in the format ###-###-####. If the data is valid, the message "Good, this is a valid phone number!" is displayed. If the data is not valid, the message "This phone number is invalid!" is displayed.

To use this script in a movie, create two text fields on the Stage and choose Input in the Text Options panel for each. Assign the variable `phoneNumber` to one text field and assign the variable `message` to the other. Attach the following action to a button on the Stage next to the text fields:

```
on (release) {
   valid = validPhoneNumber(phoneNumber);
   if (valid) {
      message = "Good, this is a valid phone number!";
   } else {
      message = "This phone number is invalid!";
   }
   function isdigit(ch) {
      return ch.length == 1 && ch >= '0' && ch <= '9';
   }
   function validPhoneNumber(phoneNumber) {
      if (phoneNumber.length != 12) {
         return false;
      }
      for (var index = 0; index < 12; index++) {
         var ch = phoneNumber.charAt(index);
         if (index == 3 || index == 7) {
            if (ch != "-") {
            return false;
            }
         } else if (!isdigit(ch)) {
            return false;
         }
      }
      return true;
   }
}
```

To send the data, create a button that has an action similar to the following: (Replace the `getURL` arguments with arguments appropriate for your movie.)

```
on (release) {
   if (valid) {
      getURL("http://www.webserver.com", "_self", "GET");
   }
}
```

For more information about these ActionScript statements, see `set`, `for`, and `if` in Chapter 7, "ActionScript Dictionary" on page 157".

Sending messages to and from the Flash Player

To send messages from a Flash movie to its host environment (for example, a Web browser, a Director movie, or the stand-alone Flash Player), you can use the `fscommand` action. This allows you to extend your movie by using the capabilities of the host. For example, you could pass an `fscommand` action to a JavaScript function in an HTML page that opens a new browser window with specific properties.

To control a movie in the 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 Flash movie. For example, you could have a link in an HTML page that sends your Flash movie to a specific frame.

Using fscommand

Use the `fscommand` action to send a message to whichever program is hosting the Flash Player. The `fscommand` action has two parameters: *`command`* and *`arguments`*. To send a message to the stand-alone version of the Flash Player, you must use predefined commands and arguments. For example, the following action sets the stand-alone player to scale the movie to the full monitor screen size when the button is released:

```
on(release){
   fscommand("fullscreen", "true");
}
```

The following table shows the values you can specify for the *command* and *arguments* parameters of the fscommand action to control a movie playing in the stand-alone player (including projectors):

Command	Arguments	Purpose
quit	None	Closes the projector.
fullscreen	true or false	Specifying true sets the 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 movie is always drawn at its original size and never scaled. Specifying true forces the movie 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 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 arguments in the *command* and *arguments* parameters. These arguments can be strings or expressions and will be used in a JavaScript function that "catches," or handles, the fscommand action.

An fscommand action invokes the JavaScript function *moviename*_DoFSCommand in the HTML page that embeds the Flash movie, where *moviename* is the name of the Flash Player as assigned by the NAME attribute of the EMBED tag or the ID attribute of the OBJECT tag. If the Flash Player is assigned the name myMovie, the JavaScript function invoked is myMovie_DoFSCommand.

To use the fscommand action to open a message box from a Flash movie in the HTML page through JavaScript:

1 In the HTML page that embeds the Flash movie, add the following JavaScript code:

```
function theMovie_DoFSCommand(command, args) {
    if (command == "messagebox") {
       alert(args);
    }
}
```

If you publish your movie using the Flash with FSCommand template in the HTML Publish Settings, this code is inserted automatically. The movie's `NAME` and `ID` attributes will be the file name. For example, for the file `myMovie.fla`, the attributes would be set to `myMovie`. For more information about publishing, see *Using Flash*.

2 In the Flash movie, add the `fscommand` action to a button:

```
fscommand("messagebox", "This is a message box invoked from
within Flash.")
```

You can also use expressions for the `fscommand` action and arguments, as in the following example:

```
fscommand("messagebox", "Hello, " & name & ", welcome to our
Web site!")
```

3 Choose File > Publish Preview > HTML to test the movie.

The `fscommand` action 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` action and carry out an action in Director. For more information, see theDirector Support Center at http://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 theFlash Support Center at http://www.macromedia.com/support/flash.

About Flash Player methods

You can use Flash Player methods to control a movie in the 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 Flash Player movies from a scripting environment other than ActionScript. Each method has a name, and most methods take arguments. An argument specifies a value that the method operates upon. The calculation performed by some methods returns a value that can be used by the scripting environment.

There are two different technologies that enable communication between the browser and the Flash Player: LiveConnect (Netscape Navigator 3.0 or later on Windows 95/98/2000/NT or Power Macintosh) and ActiveX (Microsoft Internet Explorer 3.0 and later on Windows 95/98/2000/NT). 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 the Flash Player scripting methods, use the keywords `Flash method` to search theFlash Support Center at http://www.macromedia.com/support/flash.

6

CHAPTER 6
Troubleshooting ActionScript

The level of sophistication of some actions, especially in combination with one another, can create complexity in Flash movies. As with any programming language, you can write incorrect ActionScript that causes errors in your scripts. Using good authoring techniques makes it easier to troubleshoot your movie when something behaves unexpectedly.

Flash has several tools to help you test your movies in Test-Movie Mode or in a Web browser. The Debugger shows a hierarchical display list of movie clips currently loaded in the Flash Player. It also allows you to display and modify variable values as the movie plays. In Test-Movie Mode, the Output window displays error messages and lists of variables and objects. You can also use the `trace` action in your scripts to send programming notes and values of expressions to the Output window.

Authoring and troubleshooting guidelines

If you use good authoring practices when you write scripts, your movies will have fewer bugs (programming errors). You can use the following guidelines to help prevent problems and to fix them quickly when they do occur.

Using good authoring practices

It's a good idea to save multiple versions of your movie as you work. Choose File > Save As to save a version with a different name every half hour. You can use your version history to locate when a problem began by finding the most recent file without the problem. Using this approach, you'll always have a functioning version, even if one file gets corrupted.

Another important authoring practice is to test early, test often, and test on all target platforms to find problems as soon as they develop. Choose Control > Test Movie to run your movie in test-movie mode whenever you make a significant change or before saving a version. In test-movie mode, the movie runs in a version of the stand-alone player.

If your target audience will be viewing the movie on the Web, it's important to test the movie in a browser as well. In certain situations (for example, if you're developing an intranet site) you may know the browser and platform of your target audience. If you're developing for a Web site, however, test your movie in all browsers on all potential platforms.

It's a good idea to follow these authoring practices:

- Use the `trace` action to send comments to the Output window. See "Using trace" on page 156.
- Use the `comment` action to include instructional notes that appear only in the Actions panel. See "Comments" on page 41.
- Use consistent naming conventions to identify elements in a script. For example, it's a good idea to avoid spaces in names. Start variable and function names with a lowercase letter and use a capital letter for each new word (`myVariableName`, `myFunctionName`). Start constructor function names with a capital letter (`MyConstructorFunction`). It's most important to pick a style that makes sense to you and use it consistently.
- Use meaningful variable names that reflect what kind of information a variable contains. For example, a variable containing information about the last button pressed could be called `lastButtonPressed`. A name like `foo` would make it difficult to remember what the variable contains.
- Use editable text fields in guide layers to track variable values as an alternative to using the Debugger.
- Use the Movie Explorer in Edit-Movie Mode to view the display list and view all actions in a movie. See Flash Help.
- Use the `for...in` action to loop through the properties of movie clips, including child movie clips. You can use the `for...in` action with the `trace` action to send a list of properties to the Output window. See "Repeating an action" on page 61.

Using a troubleshooting checklist

As with every scripting environment, there are certain mistakes that scripters commonly make. The following list is a good place to start troubleshooting your movie:

- Make sure you're in test-movie mode.

  Only simple button and frame actions (for example, `gotoAndPlay`, and `stop`) will work in authoring mode. Choose Control > Enable Simple Frame Actions or choose Control > Enable Simple Buttons to enable these actions.

- Make sure you do not have frame actions on multiple layers that conflict with each other.

- If you're working with the Actions panel in Normal Mode, make sure your statement is set to expression.

  If you are passing an expression in an action and haven't selected the Expression box, the value will be passed as a string. See "Using operators to manipulate values in expressions" on page 51.

- Make sure multiple ActionScript elements do not have the same name.

  It's a good idea to give every variable, function, object, and property a unique name. Local variables are exceptions, though: they only need to be unique within their scope and are often reused as counters. See "Scoping a variable" on page 48.

For more tips on troubleshooting a Flash movie, see the Flash Support Center at http://www.macromedia.com/support/flash.

Using the Debugger

The Debugger allows you to find errors in a movie as it's running in the Flash Player. You can view the display list of movie clips and loaded movies and change the values of variables and properties to determine correct values. You can then go back to your scripts and edit them so that they produce the correct results. To use the Debugger, you must run the Flash Debug Player, a special version of the Flash Player.

The Flash Debug Player installs automatically with the Flash 5 authoring application. It allows you to download the display list, variable name and value pairs, and property name and value pairs to the Debugger in the Flash authoring application.

To display the Debugger:

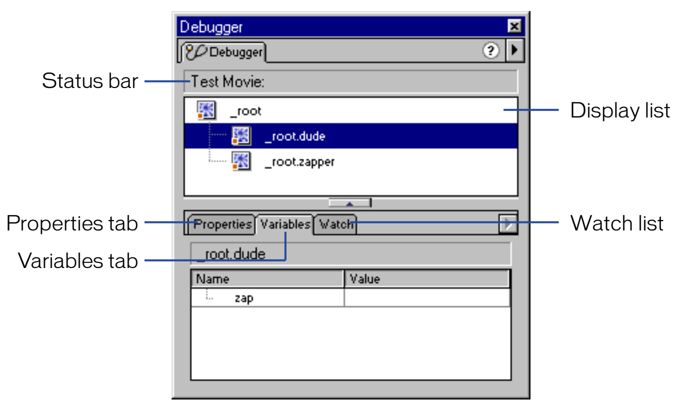
Choose Window > Debugger.

This opens the Debugger in an inactive state. No information appears in the display list until a command is issued from the Flash Player.

To activate the Debugger in test-movie mode:

Choose Control > Debug Movie.

This opens the Debugger in an active state.



Enabling debugging in a movie

When exporting a Flash Player movie, you can choose to enable debugging in your movie and create a debugging password. If you don't enable debugging, the Debugger will not activate.

As in JavaScript or HTML, any client-side ActionScript variables can potentially be viewed by the user. To store variables securely, you must send them to a server-side application instead of storing them in the movie.

However, as a Flash developer, you may have other trade secrets, such as movie clip structures, that you do not want revealed. To ensure that only trusted users can watch your movies with the Flash Debug Player, you can publish your movie with a Debugger password.

To enable debugging and create a password:

1 Choose File > Publish Settings.

2 Click the Flash tab.

3 Select Debugging Permitted.

4 To set a password, enter a password into the Password box.

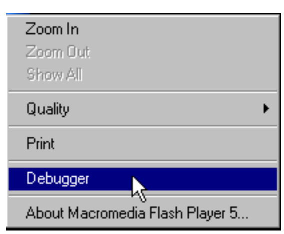
   Without this password, you cannot download information to the Debugger. If you leave the password field blank, no password is required.

To activate the Debugger in a Web browser:

1 Right-click (Windows) or Control-click (Macintosh) to open the Flash Debug Player context menu.

2 Choose Debugger.

Note: You can use the Debugger to monitor only one movie at a time. To use the Debugger, Flash must be open.



Flash Debug Player context menu

About the status bar

Once activated, the Debugger status bar displays the URL or local file path of the movie. The Flash Player is implemented in different forms depending on the playback environment. The Debugger status bar displays the type of Flash Player running the movie:

- Test-movie mode
- Stand-alone player
- Netscape plug-in

  The Netscape plug-in is used with Netscape Navigator on Windows and Macintosh and in Microsoft Internet Explorer on Macintosh.
- ActiveX control

  The ActiveX control is used with Internet Explorer on Windows.

About the display list

When the Debugger is active, it shows a live view of the movie clip display list. You can expand and collapse branches to view all movie clips currently on the Stage. When movie clips are added to or removed from the movie, the display list reflects the changes immediately. You can resize the display list by moving the horizontal splitter or by dragging from the bottom right corner.

Displaying and modifying variables

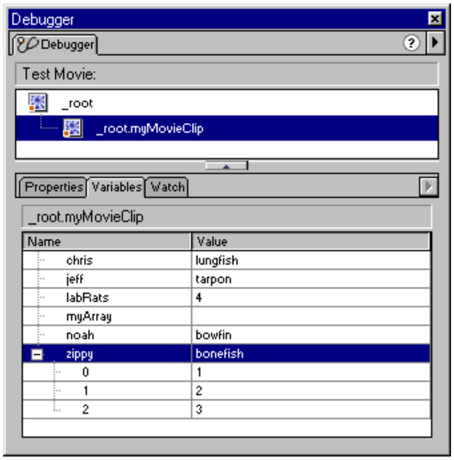
The Variables tab in the Debugger displays the names and values of any variables in the movie. If you change the value of a variable in the Variables tab, you can see the change reflected in the movie while it runs. For example, to test collision detection in a game, you could enter the variable value to position a ball in the correct location next to a wall.

To display a variable:

1 Select the movie clip containing the variable from the display list.

2 Click the Variables tab.

The display list updates automatically as the movie plays. If a movie clip is removed from the movie at a specific frame, that movie clip is also removed from the display list in the Debugger; this removes the variable name and value.



To modify a variable value:

Select the value and enter a new value.

The value must be a constant value (for example, `"Hello"`, `3523`, or `"http://www.macromedia.com"`), not an expression (for example, `x + 2`, or `eval("name:" +i)`). The value can be a string (any value surrounded by quotation marks (`""`)), a number, or a Boolean (`true` or `false`).

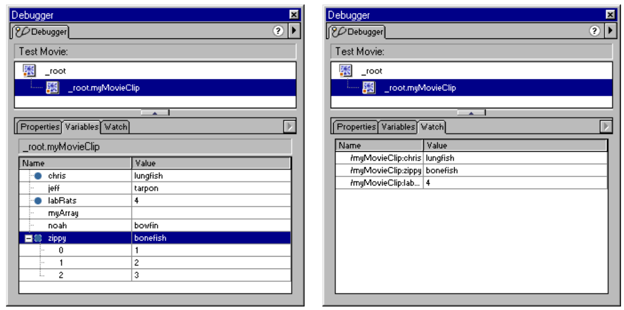
Object and Array variables are displayed in the Variables tab. Click on the Add (+) button to see their properties and values. However, you can't enter Object or Array values (for example, `{name: "I am an object"}` or `[1, 2, 3]`) in the values fields.

Note: To output the value of an expression in test-movie mode, use the `trace` action. See "Using trace" on page 156.

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 displays the absolute path to the variable and the value. You can also enter a new variable value in the Watch list.

Only variables can be added to the Watch list, 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:

- In the Variables tab, right-click (Windows) or Control-click (Macintosh) a selected variable and choose Watch from the context menu. A blue dot appears next to the variable.
- In the Watch tab, right-click (Windows) or Control-click (Macintosh) and choose Add from the context menu. Enter the variable name and value in the fields.

To remove variables from the Watch list:

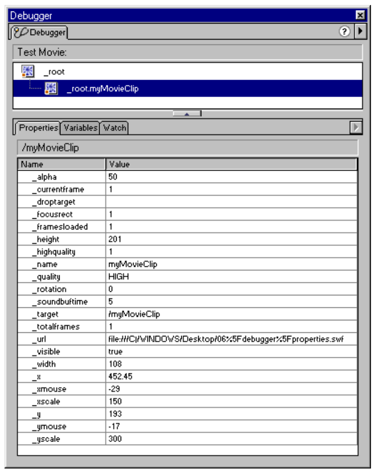
In the Watch tab, right-click (Windows) or Control-click (Macintosh) and choose Remove from the context menu.

Displaying movie properties and changing editable properties

The Debugger Properties tab displays all the property values of any movie clip on the Stage. You can change the value of a property and see the change reflected in the movie while it runs. Some movie clip properties are read-only and cannot be changed.

To display a movie clip's properties:

1. Select a movie clip from the display list.
2. Click the Properties tab.



The Properties tab in the Debugger

To modify a property value:

Select the value and enter a new value.

The value must be a constant (for example, `50`, or `"clearwater"`) rather than an expression (for example, `x + 50`). The value can be a string (any value surrounded by quotation marks (`""`)), a number, or a Boolean (`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" on page 43 and "Using operators to manipulate values in expressions" on page 51.

Note: To output the value of an expression in test-movie mode, use the `trace` action. See "Using trace" on page 156.

Using the Output window

In test-movie mode, the Output window displays information to help you troubleshoot your movie. Some information, such as syntax errors, is displayed automatically. You can display other information by using the List Objects and List Variables commands. See “Using List Objects” on page 155 and “Using List Variables” on page 155.

If you use the `trace` action in your scripts, you can send specific information to the Output window as the movie runs. This could include notes about the movie’s status or the value of an expression. See “Using trace” on page 156.

To display the Output window:

1 If your movie is not running in test-movie mode, choose Control > Test Movie.

2 Choose Window > Output.

The Output window appears.

Note: If there are syntax errors in a script, the Output window appears automatically.

3 To work with the contents of the Output window, use the Options menu:

- Choose Options > Copy to copy the contents of the Output window to the Clipboard.
- Choose Options > Clear to clear the window contents.
- Choose Options > Save to File to save the window contents to a text file.
- Choose Options > Print to print the window contents.

Using List Objects

In test-movie mode, the List Objects command displays the level, frame, object type (shape, movie clip, or button) and target path of a movie clip instance 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 movie plays; you must choose the List Objects command each time you want to send the information to the Output window.

To display a list of objects in a movie:

1 If your movie is not running in test-movie mode, choose Control > Test Movie.

2 Choose Debug > List Objects.

A list of all the objects currently on the Stage is displayed in the Output window, as in this example:

```
Layer #0: Frame=3
Movie Clip: Frame=1 Target=_root.MC
      Shape:
   Movie Clip: Frame=1 Target=_root.instance3
      Shape:
   Button:
      Movie Clip: Frame=1 Target=_root.instance3.instance2
         Shape:
```

Note: 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.

Using List Variables

In test-movie mode, the List Variables command displays a list of all the variables currently in the movie. This is especially useful for finding the correct variable target path and variable name. Unlike in the Debugger, the list does not update automatically as the movie plays; you must choose the List Variables command each time you want to send the information to the Output window.

To display a list of variable in a movie:

1 If your movie is not running in test-movie mode, choose Control > Test Movie.

2 Choose Debug > List Variables.

A list of all the variables currently in the movie is displayed in the Output window, as in this example:

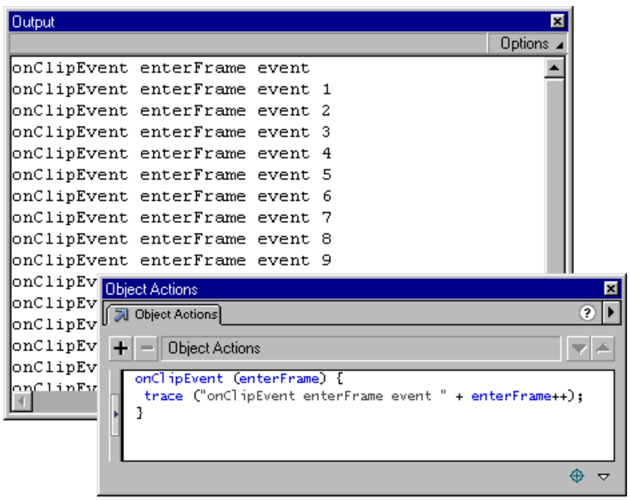
```
Level #0:
   Variable _root.country = "Sweden"
   Variable _root.city = "San Francisco"
Movie Clip:  Target=""
Variable _root.instance1.firstName = "Rick"
```

Using trace

When you use the `trace` action in a script, you can send information to the Output window. For example, while testing a movie or scene, you can send specific programming notes to the window or have specific results appear when a button is pressed or a frame is played. The `trace` action is similar to the JavaScript `alert` statement.

When you use the `trace` action in a script, you can use expressions as arguments. The value of an expression is displayed in the Output window in test-movie mode, as in the following:

```
onClipEvent(enterFrame){
   trace("onClipEvent enterFrame " + enterFrame++)
}
```



The `trace` action returns values that are displayed in the Output window.

CHAPTER 7
ActionScript Dictionary

This portion of the *ActionScript Reference Guide* describes the syntax and use of ActionScript elements in Flash 5 and later versions. The entries in this guide are the same as those in ActionScript Dictionary Help. To use examples in a script, copy the example text from ActionScript Dictionary Help and paste it in the Actions panel in Expert Mode.

The dictionary lists all ActionScript elements—operators, keywords, statements, actions, properties, functions, objects, and methods. For an overview of all dictionary entries, see Contents of the dictionary; the tables in this section are a good starting point for looking up symbolic operators or methods whose object class you don't know.

ActionScript follows the ECMA-262 standard (the specification written by the European Computer Manufacturers Association) unless otherwise noted.

There are two types of entries in this dictionary:

- Individual entries for operators, keywords, functions, variables, properties, methods, and statements
- Object entries, which provide general detail about predefined objects

Use the information in the sample entries to interpret the structure and conventions used in these two types of entries.

Sample entry for most ActionScript elements

The following sample dictionary entry explains the conventions used for all ActionScript elements that are not objects.

Entry title

All entries are listed alphabetically. The alphabetization ignores capitalization, leading underscores, and so on.

Syntax

The "Syntax" section provides correct syntax for using the ActionScript element in your code. The code portion of the syntax is in `code font`, and the arguments you must provide are in *`italicized code font`*. Brackets indicate optional arguments.

Arguments

This section describes any arguments listed in the syntax.

Description

This section identifies the element (for example, as an operator, method, function, or other element) and then describes how the element is used.

Player

This section tells which versions of the Player support the element. This is not the same as the version of Flash used to author content. For example, if you are creating content for the Flash 4 Player using the Flash 5 authoring tool, you cannot use ActionScript elements that are only available to the Flash 5 Player.

With the introduction of Flash 5 ActionScript, some Flash 4 (and earlier) ActionScript elements have been deprecated. Although deprecated elements are still supported by the Flash 5 Player, it is recommended that you use the new Flash 5 elements.

In addition, operator functionality has been greatly expanded in Flash 5. Not only have many new mathematical operators been introduced, but some of the older operators are now capable of handling additional data types. To maintain data type consistency, Flash 4 files are automatically modified when imported into the Flash 5 authoring environment, but these modifications will not affect the functionality of the original script. For more information, see the entries for + (addition), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), `!=` (inequality), and = (equality).

Example

This section provides a code sample demonstrating how to use the element.

See also

This section lists related ActionScript dictionary entries.

Sample entry for objects

The following sample dictionary entry explains the conventions used for predefined ActionScript objects. Objects are listed alphabetically with all other elements in the dictionary.

Entry title

The entry title provides the name of the object. The object name is followed by a paragraph containing general information about the object.

Method and property summary tables

Each object entry contains a table listing all of the methods associated with the object. If the object has properties (often constants), these elements are summarized in an additional table. All of the methods and properties listed in these tables also have their own dictionary entries, which follow the object entry.

Constructor

If the object requires you to use a constructor to access its methods and properties, the constructor is described at the end of the object entry. This description has all of the standard elements (syntax description, and so on) of other dictionary entries.

Method and property listings

The methods and properties of an object are listed alphabetically after the object entry.

Contents of the dictionary

All dictionary entries are listed alphabetically. However, some operators are symbols, and are presented in ASCII order. In addition, methods that are associated with an object are listed along with the object's name—for example, the `abs` method of the Math object is listed as `Math.abs`.

The following two tables will help you locate these elements. The first table lists the symbolic operators in the order in which they occur in the dictionary. The second table lists all other ActionScript elements.

Note: For precedence and associativity of operators, see Appendix A.

Symbolic operators	
--	(decrement)
++	(increment)
!	(logical NOT)
!=	(inequality)
%	(modulo)
%=	(modulo assignment)
&	(bitwise AND)
&&	(short-circuit AND)
&=	(bitwise AND assignment)
( )	(parentheses)
-	(minus)
*	(multiplication)
*=	(multiplication assignment)
,	(comma)
.	(dot)
?:	(conditional)
/	(division)
//	(comment delimiter)
/*	(comment delimiter)
/=	(division assignment)
[]	(array access)

Symbolic operators	
^	(bitwise XOR)
^=	(bitwise XOR assignment)
{}	(object initializer)
|	(bitwise OR)
||	(logical OR)
|=	(bitwise OR assignment)
~	(bitwise NOT)
+	(addition)
+=	(addition assignment)
<	(less than)
<<	(bitwise left shift)
<<=	(bitwise left shift and assignment)
<=	(less than or equal to)
<>	(inequality)
=	(assignment)
-=	(negation assignment)
==	(equality)
>	(greater than)
>=	(greater than or equal to)
>>	(bitwise right shift)
>>=	(bitwise right shift and assignment)
>>>	(bitwise unsigned right shift)
>>>=	(bitwise unsigned right shift and assignment)

The following table lists all ActionScript elements that are not symbolic operators.

ActionScript element	See entry
abs	Math.abs
acos	Math.acos
add	add
and	and
_alpha	_alpha
appendChild	XML.appendChild
Array	Array (object)
asin	Math.asin
atan	Math.atan
atan2	Math.atan2
attachMovie	MovieClip.attachMovie
attachSound	Sound.attachSound
attributes	XML.attributes
BACKSPACE	Key.BACKSPACE
Boolean	Boolean (function), Boolean (object)
break	break
call	call
CAPSLOCK	Key.CAPSLOCK
ceil	Math.ceil
charAt	String.charAt
charCodeAt	String.charCodeAt
childNodes	XML.childNodes
chr	chr
cloneNode	XML.cloneNode
close	XMLSocket.close
Color	Color (object)
concat	Array.concat, String.concat

ActionScript element	See entry
connect	XMLSocket.connect
constructor	Array, Boolean, Color, Date, Number, Object, Sound, String, XML, XMLSocket
continue	continue
CONTROL	Key.CONTROL
cos	Math.cos
createElement	XML.createElement
createTextNode	XML.createTextNode
_currentframe	_currentframe
Date	Date (object)
delete	delete
DELETEKEY	Key.DELETEKEY
docTypeDecl	XML.docTypeDecl
do...while	do...while
DOWN	Key.DOWN
_droptarget	_droptarget
duplicateMovieClip	duplicateMovieClip, MovieClip.duplicateMovieClip
E	Math.E
else	else
END	Key.END
ENTER	Key.ENTER
eq	eq (equal–string specific)
escape (function)	escape
ESCAPE (constant)	Key.ESCAPE
eval	eval
evaluate	evaluate
exp	Math.exp
firstChild	XML.firstChild
floor	Math.floor

ActionScript element	See entry
_focusrect	_focusrect
for	for
for.. in	for..in
_framesloaded	_framesloaded
fromCharCode	String.fromCharCode
fscommand	fscommand
function	function
ge	ge (greater than or equal to—string specific)
getAscii	Key.getAscii
getBeginIndex	Selection.getBeginIndex
getBounds	MovieClip.getBounds
getBytesLoaded	MovieClip.getBytesLoaded
getBytesTotal	MovieClip.getBytesTotal
getCaretIndex	Selection.getCaretIndex
getCode	Key.getCode
getDate	Date.getDate
getDay	Date.getDay
getEndIndex	Selection.getEndIndex
getFocus	Selection.getFocus
getFullYear	Date.getFullYear
getHours	Date.getHours
getMilliseconds	Date.getMilliseconds
getMinutes	Date.getMinutes
getMonth	Date.getMonth
getPan	Sound.getPan
getProperty	getProperty
getRGB	Color.setRGB
getSeconds	Date.getSeconds

ActionScript element	See entry
getTime	Date.getTime
getTimer	getTimer
getTimezoneOffset	Date.getTimezoneOffset
getTransform	Color.getTransform, Sound.getTransform
getURL	getURL, MovieClip.getURL
getUTCDate	Date.getUTCDate
getUTCDay	Date.getUTCDay
getUTCFullYear	Date.getUTCFullYear
getUTCHours	Date.getUTCHours
getUTCMilliseconds	Date.getUTCMilliseconds
getUTCMinutes	Date.getUTCMinutes
getUTCMonth	Date.getUTCMonth
getUTCSeconds	Date.getUTCSeconds
getVersion	getVersion
getVolume	Sound.getVolume
getYear	Date.getYear
globalToLocal	MovieClip.globalToLocal
gotoAndPlay	gotoAndPlay, MovieClip.gotoAndPlay
gotoAndStop	gotoAndStop, MovieClip.gotoAndStop
gt	gt (greater than –string specific)
hasChildNodes	XML.haschildNodes
_height	_height
hide	Mouse.hide
_highquality	_highquality
hitTest	MovieClip.hitTest
HOME	Key.HOME
if	if
ifFrameLoaded	ifFrameLoaded

ActionScript element	See entry
#include	#include
indexOf	String.indexOf
Infinity	Infinity
INSERT	Key.INSERT
insertBefore	XML.insertBefore
int	int
isDown	Key.isDown
isFinite	isFinite
isNaN	isNaN
isToggled	Key.isToggled
join	Array.join
Key	Key (object)
lastChild	XML.lastChild
lastIndexOf	String.indexOf
le	le (less than or equal to – string specific)
LEFT	Key.LEFT
length	length, Array.length, String.length
LN2	Math.LN2
LN10	Math.LN10
load	XML.load
loaded	XML.loaded
loadMovie	loadMovie, MovieClip.loadMovie
loadVariables	loadVariables, MovieClip.loadVariables
localToGlobal	MovieClip.localToGlobal
log	Math.log
LOG2E	Math.LOG2E
LOG10E	Math.LOG10E
lt	le (less than or equal to – string specific)

ActionScript element	See entry
Math	Math (object)
max	Math.max
maxscroll	maxscroll
MAX_VALUE	Number.MAX_VALUE
mbchr	mbchr
mblength	mblength
mbord	mbord
mbsubstring	mbsubstring
min	Math.min
MIN_VALUE	Number.MIN_VALUE
Mouse	Mouse (object)
MovieClip	MovieClip (object)
_name	_name
NaN	NaN, Number.NaN
ne	ne (not equal – string specific)
NEGATIVE_INFINITY	Number.NEGATIVE_INFINITY
new (operator)	new
newline	newline
nextFrame	nextFrame, MovieClip.nextFrame
nextScene	nextScene
nextSibling	XML.nextSibling
nodeName	XML.nodeName
nodeType	XML.nodeType
nodeValue	XML.nodeValue
not	not
null	null
Number	Number (function), Number (object)
Object	Object (object)

ActionScript element	See entry
On	on(mouseEvent)
onClipEvent	onClipEvent
onClose	XMLSocket.onClose
onConnect	XMLSocket.onConnect
OnLoad	XML.onLoad
onXML	XMLSocket.onXML
or (logical OR)	or
ord	ord
_parent	_parent
parentNode	XML.parentNode
parseFloat	parseFloat
parseInt	parseInt
parseXML	XML.parseXML
PGDN	Key.PGDN
PGUP	Key.PGUP
PI	Math.PI
play	play, MovieClip.play
pop	Array.pop
POSITIVE_INFINITY	Number.POSITIVE_INFINITY
pow	Math.pow
prevFrame	prevFrame, MovieClip.prevFrame
previousSibling	XML.previousSibling
prevScene	prevScene
print	print
printAsBitmap	printAsBitmap
push	Array.push
_quality	_quality
random	random, Math.random

ActionScript element	See entry
removeMovieClip	removeMovieClip, MovieClip.removeMovieClip
removeNode	XML.removeNode
return	return
reverse	Array.reverse
RIGHT	Key.RIGHT
_root	_root
_rotation	_rotation
round	Math.round
scroll	scroll
Selection	Selection (object)
send	XML.send, XMLSocket.send
sendAndLoad	XML.sendAndLoad
set	set
setDate	Date.setDate
setFocus	Selection.setFocus
setFullYear	Date.setFullYear
setHours	Date.setHours
setMilliseconds	Date.setMilliseconds
setMinutes	Date.setMinutes
setMonth	Date.setMonth
setPan	Sound.setPan
setProperty	setProperty
setRGB	Color.setRGB
setSeconds	Date.setSeconds
setSelection	Selection.setSelection
setTime	Date.setTime
setTransform	Color.setTransform,Sound.setTransform
setUTCDate	Date.setUTCDate

ActionScript element	See entry
setUTCFullYear	Date.setUTCFullYear
setUTCHours	Date.setUTCHours
setUTCMilliseconds	Date.setUTCMilliseconds
setUTCMinutes	Date.setUTCMinutes
setUTCMonth	Date.setUTCMonth
setUTCSeconds	Date.setUTCSeconds
setVolume	Sound.setVolume
setYear	Date.setYear
shift (method)	Array.shift
SHIFT (constant)	Key.SHIFT
show	Mouse.show
sin	Math.sin
slice	Array.slice, String.slice
sort	Array.sort
Sound	Sound (object)
_soundbuftime	_soundbuftime
SPACE	Key.SPACE
splice	Array.splice
split	String.split
sqrt	Math.sqrt
SQRT1_2	Math.SQRT1_2
SQRT2	Math.SQRT2
start	Sound.start
startDrag	startDrag, MovieClip.startDrag
status	XML.status
stop	stop,MovieClip.stop,Sound.stop
stopAllSounds	stopAllSounds
stopDrag	stopDrag, MovieClip.stopDrag

ActionScript element	See entry
String	String (function), String (object)," " (string delimiter)
substr	String.substr
substring	substring, String.substring
swapDepths	MovieClip.swapDepths
TAB	Key.TAB
tan	Math.tan
_target	_target
targetPath	targetPath
tellTarget	tellTarget
this	this
toggleHighQuality	toggleHighQuality
toLowerCase	String.toLowerCase
toString	Array.toString,Boolean.toString, Date.toString, Number.toString, Object.toString, XML.toString
_totalframes	_totalframes
toUpperCase	String.toUpperCase
trace	trace
typeof	typeof
unescape	unescape
unloadMovie	unloadMovie, MovieClip.unloadMovie
unshift	Array.shift
UP	Key.UP
updateAfterEvent	updateAfterEvent
_url	_url
UTC	Date.UTC
valueOf	Boolean.valueOf, Number.valueOf, Object.valueOf
var	var
_visible	_visible
void	void

ActionScript element	See entry
while	while
_width	_width
with	with
_x	_x
XML	XML (object)
xmlDecl	XML.xmlDecl
XMLSocket	XMLSocket (object)
_xmouse	_xmouse
_xscale	_xscale
_y	_y
_ymouse	_ymouse
_yscale	_yscale

-- (decrement)

Syntax

--expression
expression--

Arguments

expression A variable, number, element in an array, or property of an object.

Description

Operator; a pre-decrement and post-decrement unary operator that subtracts 1 from the *expression*. The pre-decrement form of the operator (*--expression*) subtracts 1 from *expression* and returns the result. The post-decrement form of the operator (*expression--*) subtracts 1 from the *expression* and returns the initial value of the *expression* (the result prior to the subtraction).

Player

Flash 4 or later.

Example

The pre-decrement form of the operator decrements `x` to 2 (`x - 1 = 2`), and returns the result as `y`:

```
x = 3;
```

```
y = --x
```

The post-decrement form of the operator decrements `x` to 2 (`x - 1 = 2`), and returns the original value (`x = 3`) as the result `y`:

If `x = 3;`

```
y = x--
```

++ (increment)

Syntax

`++expression`
`expression++`

Arguments

`expression` A variable, number, element in an array, or property of an object.

Description

Operator; a pre-increment and post-increment unary operator that adds 1 to the `expression`. The pre-increment form of the operator (`++expression`) adds 1 to the `expression` and returns the result. The post-increment form of the operator (`expression++`) adds 1 to the `expression` and returns the initial value of the `expression` (the result prior to the addition).

The pre-increment form of the operator increments `x` to 2 (`x + 1 = 2`), and returns the result as `y`:

```
x = 1;
y = ++x
```

The post-increment form of the operator increments `x` to 2 (`x + 1 = 2`), and returns the original value (`x = 1`) as the result `y`:

```
x = 1;
y = x++;
```

Player

Flash 4 or later.

Example

The following example uses ++ as a pre-increment operator with a `while` statement.

```
i = 0
while(i++ < 5){
// this section will execute five times
}
```

The following example uses ++ as a pre-increment operator:

```
var a = [];
var i = 0;
while (i < 10) {
   a.push(++i);
}
trace(a.join());
```

This script prints the following:

```
1,2,3,4,5,6,7,8,9
```

The following example uses ++ as a post-increment operator:

```
var a = [];
var i = 0;
while (i < 10) {
a.push(i++);
   }
trace(a.join());
```

This script prints the following:

```
0,1,2,3,4,5,6,7,8,9
```

! (logical NOT)

Syntax

`!`*`expression`*

Arguments

`expression` A variable or evaluated expression.

Description

Operator (logical); inverts the Boolean value of a variable or expression. If *`expression`* is a variable with an absolute or converted value `true`, `!`*`variable`* the value of `! expression` is `false`. If the expression `x && y` evaluates to `false`, the expression `!(x && y)` evaluates to `true`. This operator is identical to the `not` operator that was used in Flash 4.

Player

Flash 4 or later.

Example

In the following example the variable `happy` is set to `false`, the `if` condition evaluates the condition `!happy`, and if the condition is `true`, `trace` sends a string to the Output window.

```
happy = false;
if (!happy){
trace("don't worry be happy");
}
```

The following illustrates the results of the`!` operator:

`!` *`true`* returns `false`

`!` *`false`* returns `true`

!= (inequality)

Syntax

expression1 != *expression2*

Arguments

expression1, expression2 Numbers, strings, Booleans, variables, objects, arrays, or functions.

Description

Operator (equality); tests for the exact opposite of the == operator. If *expression1* is equal to *expression2*, the result is `false`. As with the == operator, the definition of *equal* depends on the data types being compared.

- Numbers, strings, and Boolean values are compared by value.
- Variables, objects, arrays, and functions are compared by reference.

Player

Flash 5 or later.

Example

The following example illustrates the results of the != operator.

`5 != 8` returns `true`

`5 != 5` returns `false`

The following example illustrates the use of the != operator in an `if` statement:

```
a = "David";
b = "Fool"
if (a != b){
trace("David is not a fool");
}
```

See also

`== (equality)`

% (modulo)

Syntax

`expression1 % expression2`

Arguments

`expression1, expression2` Numbers, integers, floating-point numbers, or strings that convert to a numeric value.

Description

Operator (arithmetic); calculates the remainder of *`expression1`* divided by *`expression2`*. If either of the *`expression`* arguments are nonnumeric, the modulo operator attempts to convert them to numbers.

Player

Flash 4 or later. In Flash 4 files, the `%` operator is expanded in the SWF file as `x - int(x/y) * y`, and may not be as fast or as accurate as the Flash 5 Player implementation.

Example

The following is a numeric example of using the `%` operator:

`12 % 5` returns `2`

`4.3 % 2.1` returns `0.1`

%= (modulo assignment)

Syntax

expression1 %= *expression2*

Arguments

expression1,*expression2* Integers and variables.

Description

Operator (assignment); assigns *expression1* the value of *expression1* % *expression2*.

Player

Flash 4 or later.

Example

The following illustrates using the `%=` operator with variables and numbers:

`x %= y` is the same as `x = x % y`

If `x = 14` and `y = 5` then

`x %= 5` returns `4`

See also

`% (modulo)`

& (bitwise AND)

Syntax

expression1 & *expression2*

Arguments

expression1, expression2 Any number.

Description

Operator (bitwise); converts *expression1* and *expression2* to 32-bit unsigned integers, and performs a Boolean AND operation on each bit of the integer arguments. The result is a new 32-bit unsigned integer.

Player

Flash 5 or later. In Flash 4 the `&` operator was used for concatenating strings. In Flash 5 the `&` operator is a bitwise AND, and the `add` and `+` operators concatenate strings. Flash 4 files that use the `&` operator are automatically updated to use `add` when brought into the Flash 5 authoring environment.

&& (short-circuit AND)

Syntax

expression1 `&&` *expression2*

Arguments

expression1, expression2 Numbers, strings, variables, or functions.

Description

Operator (logical); performs a Boolean operation on the values of one or both of the expressions. Causes the Flash interpreter to evaluate *expression1* (the left expression) and returns `false` if the expression evaluates to `false`. If *expression1* evaluates to `true`, *expression2* (the right) is evaluated. If *expression2* evaluates to `true`, the final result is `true`; otherwise, it is `false`.

Player

Flash 4 or later.

Example

This example assigns the values of the evaluated expressions to the variables `winner` and `loser` in order to perform a test:

```
winner = (chocolateEggs >=10) && (jellyBeans >=25);
loser = (chocolateEggs <=1) && (jellyBeans <= 5);
if (winner) {
   alert = "You Win the Hunt!";
   if (loser) {
      alert = "Now THAT'S Unhappy Hunting!";
   }
} else {
   alert = "We're all winners!";
}
```

&= (bitwise AND assignment)

Syntax

`expression1` `&=` *`expression2`*

Arguments

`expression1, expression2` Integers and variables.

Description

Operator (bitwise assignment); assigns *`expression1`* the value of *`expression1`* `&` *`expression2`*.

Player

Flash 5 or later.

Example

The following illustrates using the `&=` operator with variables and numbers:

`x &= y` is the same as `x = x & y`

If `x = 15` and `y = 9` then

`x &= 9` returns `9`

See also

`& (bitwise AND)`

() (parentheses)

Syntax

```
(expression1, expression2);
function(functionCall1, ..., functionCallN);
```

Arguments

expression1, expression2 Numbers, strings, variables, or text.

function The function to be performed on the contents of the parentheses.

functionCall1...functionCallN A series of functions to execute before the result is passed to the function outside the parentheses.

Description

Operator (general); performs a grouping operation on one or more arguments, or surrounds one or more arguments and passes the results a parameter to a function outside the parentheses.

Usage 1: Performs a grouping operation on one or more expressions to control the order of execution of the operators in the expression. This operator overrides the automatic precedence order, and causes the expressions within the parentheses to be evaluated first. When parentheses are nested, Flash evaluates the contents of the innermost parentheses before the contents of the outer ones.

Usage 2: Surrounds one or more arguments and passes them as parameters to the function outside the parentheses.

Player

Flash 4 or later.

Example

(Usage 1) The following statements illustrate the use of parentheses to control the order of execution of expressions. (The result appears below each statement.)

```
(2 + 3) * (4 + 5)
45
2 + (3 * (4 + 5))
29
2 + (3 * 4) + 5
19
```

(Usage 2) The following example illustrates the use of parentheses with a function:

```
getDate();
invoice(item, amount);
```

See also

`with`

– (minus)

Syntax

(Negation) –*expression*

(Subtraction) *expression1* - *expression2*

Arguments

expression1, expression2 Any number.

Description

Operator (arithmetic); used for negating or subtracting. When used for negating, it reverses the sign of the numerical *expression*. When used for subtracting, it performs an arithmetic subtraction on two numerical expressions, subtracting *expression2* from *expression1*. When both expressions are integers, the difference is an integer. When either or both expressions are floating-point numbers, the difference is a floating-point number.

Player

Flash 4 or later.

Example

(Negation) This statement reverses the sign of the expression 2 + 3:

```
-(2 + 3)
```

The result is -5.

(Subtraction) This statement subtracts the integer 2 from the integer 5:

```
5 - 2
```

The result is 3, which is an integer.

(Subtraction): This statement subtracts the floating-point number 1.5 from the floating-point number 3.25:

```
put 3.25 - 1.5
```

The result is 1.75, which is a floating-point number.

* (multiplication)

Syntax

`expression1 * expression2`

Arguments

`expression1, expression2` Integers or floating-point numbers.

Description

Operator (arithmetic); multiplies two numerical expressions. If both expressions are integers, the product is an integer. If either or both expressions are floating-point numbers, the product is a floating-point number.

Player

Flash 4 or later.

Example

This statement multiplies the integers 2 and 3:

```
2 * 3
```

The result is 6, which is an integer.

Example

This statement multiplies the floating-point numbers 2.0 and 3.1416:

```
2.0 * 3.1416
```

The result is 6.2832, which is a floating-point number.

*= (multiplication assignment)

Syntax

expression1 *= *expression2*

Arguments

expression1, expression2 Integers, floating-point numbers, or strings.

Description

Operator (assignment); assigns *expression1* the value of *expression1* * *expression2*.

Player

Flash 4 or later.

Example

The following illustrates using the *= operator with variables and numbers:

`x *= y` is the same as `x = x * y`

If `x = 5` and `y = 10` then

`x *= 10` returns `50`

See also

`* (multiplication)`

, (comma)

Syntax

expression1, *expression2*

Arguments

expression Any number, variable, string, array element, or other data.

Description

Operator; instructs Flash to evaluate *expression1*, then *expression2*, and return the value of *expression2*. This operator is primarily used with the `for` loop statement.

Player

Flash 4 or later.

Example

The following code sample uses the comma operator:

```
var a=1, b=2, c=3;
```

This is equivalent to writing the following:

```
var a=1;
var b=2;
var c=3;
```

. (dot operator)

Syntax

`object.property_or_method`
`instancename.variable`
`instancename.childinstance.variable`

Arguments

`object` An instance of an object. Some objects require that instances be created using the constructor for that object. The object can be any of the predefined ActionScript objects or a custom object. This argument is always to the left of the dot (.) operator.

`property_or_method` The name of a property or method associated with an object. All of the valid method and properties for the predefined objects are listed in the Method and Property summary tables for that object. This argument is always to the right of the dot (.) operator.

`instancename` The name of a movie clip instance.

`childinstance` An movie clip instance that is a child of the main movie clip.

`variable` A variable in a movie clip.

Description

Operator; used to navigate movie clip hierarchies in order to access nested child movie clips, variables, or properties. The dot operator is also used to test or set the properties of an object, execute a method of an object, or create a data structure.

Player

Flash 4 or later.

See also

`[] (array access operator)`

Example

This statement identifies the current value of the variable `hairColor` by the movie clip `person`:

```
person.hairColor
```

This is equivalent to the following Flash 4 syntax:

```
/person:hairColor
```

Example

The following code illustrates how the dot operator can be used to create a structure of an array:

```
account.name = "Gary Smith";
account.address = "123 Main St ";
account.city = "Any Town";
account.state = "CA";
account.zip = "12345";
```

?: (conditional)

Syntax

expression1 ? *expression2* : *expression3*

Arguments

expression1 An expression that evaluates to a Boolean value, usually a comparison expression.

expression2, *expression3* Values of any type.

Description

Operator (conditional); instructs Flash to evaluate *expression1*, and return the value of *expression2* if *expression1* is `true`; otherwise return the value of the *expression3*.

Player

Flash 4 or later.

/ (division)

Syntax

expression1 / *expression2*

Arguments

expression Any number.

Description

Operator (arithmetic); divides *expression1* by *expression2*. The expression arguments and results of the division operation are treated and expressed as double-precision floating-point numbers.

Player

Flash 4 or later.

Example

This statement divides the floating-point number 22.0 by 7.0 and then displays the result in the Output window:

```
trace(22.0 / 7.0);
```

The result is 3.1429, which is a floating-point number.

// (comment delimiter)

Syntax

```
// comment
```

Arguments

comment Text that is not part of the code, and should be ignored by the interpreter.

Description

Comment; indicates the beginning of a script comment. Any text that appears between the comment delimiter `//` and the end-of-line character is interpreted as a comment and ignored by the ActionScript interpreter.

Player

Flash 1 or later.

Example

This script uses comment delimiters slash to identify the first, third, fifth, and seventh lines as comments:

```
// set the X position of the ball movie clip
ball = getProperty(ball._x);
// set the Y position of the ball movie clip
ball = getProperty(ball._y);
// set the X position of the kitty movie clip
kitty = getProperty(kitty._x);
// set the Y position of the kitty movie clip
kitty_y = getProperty(kitty._y);
```

See also

`/* (comment delimiter)`

/* (comment delimiter)

Syntax

```
/* comment */
/*
* comment
* comment
*/
```

Arguments

comment Any text

Description

Comment; indicates one or more lines of script comments. Any text that appears between the opening comment tag `/*` and the closing comment tag `*/`, is interpreted as a comment and ignored by the ActionScript interpreter. Use the first syntax to identify single-line comments, and use the second syntax to identify comments on multiple successive lines. Leaving off the closing tag `*/` when using this form of comment delimiter causes the ActionScript compiler to return an error message.

Player

Flash 5 or later.

See also

`// (comment delimiter)`

/= (division assignment)

Syntax

```
expression1 /= expression2
```

Arguments

expression1,expression2 Integers, floating-point numbers, or strings.

Description

Operator (assignment); assigns *expression1* the value of *expression1 / expression2*.

Player

Flash 4 or later.

Example

The following illustrates using the `/=` operator with variables and numbers:

```
x /= y is the same as x = x /y
x = 10;
y = 2;
x /= y;
// x now contains the value 5
```

[] (array access operator)

Syntax

```
myArray["a0", "a1",..."aN"];
object[value1, value2, ...valueN];
```

Arguments

myArray The name of an array.

a0, a1,...aN Elements in an array.

value1, 2,...N Names of properties.

Description

Operator; creates a new object initializing the properties specified in the arguments, or initializes new array with the elements (*a0*) specified in the arguments.

The created object has the generic Object object as its prototype. Using this operator is the same as calling `new Object` and populating the properties using the assignment operator. Using this operator is an alternative to using the `new` operator, which allows for the quick and convenient creation of objects.

Player

Flash 4 or later.

Example

The following example code samples are two different ways of creating a new empty Array object:

```
myArray =[];
myArray = new Array();
```

The following is an example of a simple array:

```
myArray = ["red", "orange", "yellow", "green", "blue", "purple"]
myArray[0]="red"
myArray[1]="yellow"
myArray[2]="green"
myArray[3]="blue"
myArray[4]="purple"
```

^(bitwise XOR)

Syntax

expression1 ^ *expression2*

Arguments

expression1,expression2 Any number.

Description

Operator (bitwise); converts *expression1* and *expression2* to 32-bit unsigned integers, and returns a 1 in each bit position where the corresponding bits in *expression1* or *expression1*, but not both, are 1.

Player

Flash 5 or later.

Example

```
15 ^ 9 returns 6
(1111 ^ 1001 = 0110)
```

^= (bitwise XOR assignment)

Syntax

expression1 ^= *expression2*

Arguments

expression1,expression2 Integers and variables.

Description

Operator (compound assignment); assigns *expression1* the value of *expression1* ^ *expression2*.

Player

Flash 5 or later.

Example

The following is an example of a ^= operation:

```
// 15 decimal = 1111 binary
x = 15;
// 9 decimal = 1001 binary
x ^= y;
returns
x ^ y (0110 binary)
```

The following illustrates using the ^= operator with variables and numbers:

```
x ^= y is the same as x = x ^ y
If x = 15 and y = 9 then
15 ^= 9 returns 6
```

See also

`^(bitwise XOR)`

{} (object initializer)

Syntax

```
object {name1: value1,
   name1: value2,
...
   nameN: valueN };
```

Arguments

`object` The object to create.

`name1,2,...N` The name of the property.

`value1,2,...N` The corresponding value for each *`name`* property.

Description

Operator; creates a new object and initializes it with the specified *`name`* and *`value`* property pairs. The created object has the generic Object object as its prototype. Using this operator is the same as calling `new Object` and populating the property pairs using the assignment operator. Using this operator is an alternative to using the `new` operator, which allows for the quick and convenient creation of objects.

Player

Flash 5 or later.

Example

The following code shows how an empty object can be created using the object initializer operator and using the `new Object`:

```
object = {};
object = new Object();
```

The following creates an object `account` initializing the properties `name`, `address`, `city`, `state`, `zip`, and `balance`:

```
account = { name: "John Smith",
   address: "123 Main Street",
   city: "Blossomville",
   state: "California",
   zip: "12345",
   balance: "1000" };
```

The following example shows how array and object initializers can be nested within each other:

```
person = { name: "Peter Piper",
   children: [ "Jack", "Jill", "Moe",] };
```

The following example is another way of using the information in the previous example above, with the same results:

```
person = new Person();
person.name = 'John Smith';
person.children = new Array();
person.children[0] = 'Jack';
person.children[1] = 'Jill';
person.children[2] = 'Moe';
```

See also

`[] (array access operator)`
`new`
`Object (object)`

| (bitwise OR)

Syntax

expression1 | *expression2*

Arguments

expression1,*expression2* Any number.

Description

Operator (bitwise); converts *expression1* and *expression2* to 32-bit unsigned integers, and returns a 1 in each bit position where the corresponding bits of either *expression1* or *expression2* are 1.

Player

Flash 5 or later.

Example

The following is an example of a bitwise OR operation. Note that 15 is 1111 binary:

```
// 15 decimal = 1111 binary
x = 15;
// 9 decimal = 1001 binary
y = 9;
// x | y = binary
z = x | y;
z = 15
```

The following is another way of expressing the preceding example:

```
15 | 9 returns 15
(1111 | 0011 = 1111)
```

|| (OR)

Syntax

expression1 || *expression2*

Arguments

expression1,expression2 A Boolean value or expression that converts to a Boolean value.

Description

Operator (logical); evaluates *expression1* and *expression2*. The result is (`true`) if either or both expressions evaluate to `true`; the result is (`false`) only if both expressions evaluate to `false`.

With non-Boolean expressions, the logical OR operator causes Flash to evaluate the expression on the left; if it can be converted to `true`, the result is `true`. Otherwise, it evaluates the expression on the right and the result is the value of that expression.

Player

Flash 4 or later.

Example

The following example uses the `||` operator in an `if` statement:

```
want = true;
need = true;
love = false;
if (want || need || love){
trace("two out of 3 ain't bad");
}
```

|= (bitwise OR assignment)

Syntax

expression1 `|=` *expression2*

Arguments

expression1,expression2 Integers and variables.

Description

Operator (assignment); assigns *expression1* the value of *expression1* `|` *expression2*.

Player

Flash 5 or later.

Example

The following illustrates using the `|=` operator with variables and numbers:

`x |= y` is the same as `x = x | y`

If `x = 15` and `y = 9` then

`x |= 9` returns `15`

See also

`| (bitwise OR)`

~ (bitwise NOT)

Syntax

~ *expression*

Arguments

expression Any number.

Description

Operator (bitwise); converts the *expression* to a 32-bit unsigned integer, then inverts the bits. Or, simply said, changes the sign of a number and subtracts 1.

A bitwise NOT operation changes the sign of a number and subtracts 1.

Player

Flash 5 or later.

Example

The following is a numerical explanation of a bitwise NOT operation performed on a variable:

`~a`, returns `-1` if `a = 0`, and returns `-2` if `a = 1`, thus:

`~0=-1` and `~1=-2`

+ (addition)

Syntax

```
expression1 + expression2
```

Arguments

expression1,expression2 Integers, numbers, floating-point numbers, or strings.

Description

Operator; adds numeric expressions or concatenates strings. If one expression is a string, all other expressions are converted to strings and concatenated.

If both expressions are integers, the sum is an integer; if either or both expressions are floating-point numbers, the sum is a floating-point number.

Player

Flash 4; Flash 5 or later. In Flash 5, + is a numeric operator or string concatenator depending on the data type of the argument. In Flash 4, + is only a numeric operator. Flash 4 files brought into the Flash 5 authoring environment undergo a conversion process to maintain data type integrity. The first example below illustrates the conversion process.

Example

The following illustrates the conversion of a Flash 4 file containing a numeric quality comparison:

Flash 4 file:

```
x + y
```

Converted Flash 5 file:

```
Number(x) + Number(y)
```

This statement adds the integers 2 and 3 and then displays the resulting integer, 5, in the Output window:

```
trace (2 + 3);
```

This statement adds the floating-point numbers 2.5 and 3.25 and displays the result, 5.7500, a floating-point number, in the Output window:

```
trace (2.5 + 3.25);
```

This statement concatenates two strings and displays the result, “today is my birthday,” in the Output window:

```
"today is my" + "birthday"
```

See also

`add`

+= (addition assignment)

Syntax

expression1 `+=` *expression2*

Arguments

expression1,*expression2*   Integers, floating-point numbers, or strings.

Description

Operator (compound assignment); assigns *expression1* the value of *expression1* `+` *expression2*. This operator also performs string concatenation.

Player

Flash 4 or later.

Example

This following illustrates a numeric use of the += operator:

`x += y` is the same as `x = x + y`

If x = 5 and y = 10 then

`x += 10` returns 15

This example illustrates using the += operator with a string expression:

```
x = "My name is"
x += "Mary"
```

The result for the above code is as follows:

```
"My name is Mary"
```

See also

`+ (addition)`

< (less than)

Syntax

`expression1 < expression2`

Arguments

`expression1,expression2` Numbers or strings.

Description

Operator (comparison); compares two expressions and determines whether *`expression1`* is less than *`expression2`* (`true`), or whether *`expression1`* is greater than or equal to *`expression2`* (`false`). String expressions are evaluated and compared based on the number of characters in the string.

Player

Flash 4; Flash 5 or later. In Flash 5 `<` is a comparison operator capable of handling various data types. In Flash 4 `<` is an numeric operator. Flash 4 files brought into the Flash 5 authoring environment undergo a conversion process to maintain data type integrity. The first example below illustrates the conversion process.

Example

The following illustrates the conversion of a Flash 4 file containing a numeric quality comparison.

Flash 4 file:

```
x < y
```

Converted Flash 5 file:

```
Number(x) < Number(y)
```

The following examples illustrate `true` and `false` returns for both numbers and strings:

`3 < 10` or `"Al" < "Jack"` return `true`

`10 < 3` or `"Jack" < "Al"` return `false`

<< (bitwise left shift)

Syntax

expression1 << *expression2*

Arguments

expression1 A number, string, or expression to be shifted left.

expression2 A number, string, or expression that converts to an integer from 0 to 31.

Description

Operator (bitwise); converts *expression1* and *expression2* to 32-bit integers, and shifts all of the bits in *expression1* to the left by the number of places specified by the integer resulting from the conversion of *expression2*. The bit positions that are emptied as a result of this operation are filled in with 0. Shifting a value left by one position is the equivalent of multiplying it by 2.

Player

Flash 5 or later.

Example

The following example shifts the integer `1` ten bits to the left:

```
x = 1 << 10
```

The result of this operation is `x = 1024`. This is because 1 decimal equals 1 binary, 1 binary shifted left by 10 is 10000000000 binary, and 10000000000 binary is 1024 decimal.

This following example shifts the integer `7` eight bits to the left:

```
x = 7 << 8
```

The result of this operation is `x = 1792`. This is because 7 decimal equals 111 binary, 111 binary shifted left by 8 bits is 11100000000 binary, and 11100000000 binary is 1792 decimal.

See also

`>>= (bitwise right shift and assignment)`

<<= (bitwise left shift and assignment)

Syntax

expression1 <<= *expression2*

Arguments

expression1 A number, string, or expression to be shifted left.

expression2 A number, string, or expression that converts to an integer from 0 to 31.

Description

Operator (compound assignment); this operator performs a bitwise left shift operation and stores the contents as a result in *expression1*.

Player

Flash 5 or later.

Example

The following two expressions are equivalent:

```
A <<= B
A = (A << B)
```

See also

```
<< (bitwise left shift)
>>= (bitwise right shift and assignment)
```

<= (less than or equal to)

Syntax

expression1 `<=` *expression2*

Arguments

expression1,expression2 Number or strings.

Description

Operator (comparison); compares two expressions and determines whether *expression1* is less than or equal to *expression2* (`true`), or whether *expression1* is greater than *expression2* (`false`).

Player

Flash 4; Flash 5 or later. In Flash 5 `<=` is a comparison operator capable of handling various data types. In Flash 4 `<=` is an numeric operator. Flash 4 files brought into the Flash 5 authoring environment undergo a conversion process to maintain data type integrity. The first example below illustrates the conversion process.

Example

The following illustrates the conversion of a Flash 4 file containing a numeric quality comparison.

Flash 4 file:

```
x <= y
```

Converted Flash 5 file:

```
Number(x) <= Number(y)
```

The following examples illustrate `true` and `false` results for both numbers and strings:

`5 <= 10` or `"Al" <= "Jack"` returns `true`

`10<= 5` or `"Jack" <= "Al"` returns `false`

<> (inequality)

Syntax

`expression1 <> expression2`

Arguments

`expression1,expression2` Numbers, strings, Booleans, variables, objects, arrays, or functions.

Description

Operator (equality); tests for the exact opposite of the == operator. If *`expression1`* is equal to *`expression2`*, the result is `false`. As with the == operator, the definition of *equal* depends on the data types being compared:

- Numbers, strings, and Boolean values are compared by value.
- Variables, objects, arrays, and functions are compared by reference.

This operator has been deprecated in Flash 5, and users are encouraged to make use of the new != operator.

Player

Flash 2 or later.

See also

`!= (inequality)`

= (assignment)

Syntax

expression1 = *expression2*

Arguments

expression1 A variable, element of an array, or property of an object.

expression2 A value of any type.

Description

Operator (assignment); assigns the type of *expression2* (the argument on the right) to the variable, array element, or property in *expression1*.

Player

Flash 4; Flash 5 or later. In Flash 5 = is an assignment operator and the == operator is used to evaluate equality. In Flash 4 = is a numeric equality operator. Flash 4 files brought into the Flash 5 authoring environment undergo a conversion process to maintain data type integrity. The first example below illustrates the conversion process.

Example

The following illustrates the conversion of a Flash 4 file containing a numeric quality comparison.

Flash 4 file:

```
x = y
```

Converted Flash 5 file:

```
Number(x) == Number(y)
```

The following example uses the assignment operator to assign the number data type to the variable `x`:

```
x = 5
```

The following example uses the assignment operator to assign the string data type to the variable `x`:

```
x = "hello"
```

-= (negation assignment)

Syntax

`expression1 -= expression2`

Arguments

`expression1,expression2` Integers, floating-point numbers, or strings.

Description

Operator (compound assignment); assigns *`expression1`* the value of *`expression1 - expression2`*.

Player

Flash 4 or later.

Example

The following illustrates using the `-=` operator with variables and numbers:

`x -= y` is the same as `x = x - y`

If `x = 5` and `y = 10` then

`x -= 10` returns `-5`

== (equality)

Syntax

`expression1 == expression2`

Arguments

`expression1,expression2` Numbers, strings, Booleans, variables, objects, arrays, or functions.

Description

Operator (equality); tests two expressions for equality. The result is `true` if the expressions are equal.

The definition of *equal* depends on the data type of the argument:

- Numbers, strings, and Boolean values are compared by value, and are considered equal if they have the same value. For instance, two strings are equal if they have the same number of characters.
- Variables, objects, arrays, and functions are compared by reference. Two variables are equal if they refer to the same object, array, or function. Two separate arrays are never considered equal, even if they have the same number of elements.

Player

Flash 5 or later.

Example

The following example uses the == operator with an `if` statement:

```
a = "David" , b = "David";
if (a == b){
trace("David is David");
}
```

> (greater than)

Syntax

`expression1 > expression2`

Arguments

`expression1,expression2` Integers, floating-point numbers, or strings.

Description

Operator (comparison); compares two expressions and determines whether *`expression1`* is greater than *`expression2`* (`true`), or whether *`expression1`* is less than or equal to *`expression2`* (`false`).

Player

Flash 4; Flash 5 or later. In Flash 5 > is a comparison operator capable of handling various data types. In Flash 4 > is an numeric operator. Flash 4 files brought into the Flash 5 authoring environment undergo a conversion process to maintain data type integrity. The example below illustrates the conversion process.

Example

The following illustrates the conversion of a Flash 4 file containing a numeric quality comparison.

Flash 4 file:

```
x > y
```

Converted Flash 5 file:

```
Number(x) > Number(y)
```

>= (greater than or equal to)

Syntax

expression1 >= *expression2*

Arguments

expression1, expression2 Strings, integers, or floating-point numbers.

Description

Operator (comparison); compares two expressions and determines whether *expression1* is greater than or equal to *expression2* (`true`), or whether *expression1* is less than *expression2* (`false`).

Player

Flash 4; Flash 5 or later. In Flash 5 `>=` is a comparison operator capable of handling various data types. In Flash 4 `>=` is a numeric operator. Flash 4 files brought into the Flash 5 authoring environment undergo a conversion process to maintain data type integrity. The example below illustrates the conversion process.

Example

The following illustrates the conversion of a Flash 4 file containing a numeric quality comparison.

Flash 4 file:

```
x >= y
```

Converted Flash 5 file:

```
Number(x) >= Number(y)
```

>> (bitwise right shift)

Syntax

`expression1 >> expression2`

Arguments

`expression1` A number, string, or expression to be shifted right.

`expression2` A number, string, or expression that converts to an integer from 0 to 31.

Description

Operator (bitwise); converts `expression1` and `expression2` to 32-bit integers, and shifts all of the bits in `expression1` to the right by the number of places specified by the integer resulting from the conversion of `expression2`. Bits that are shifted off to the right are discarded. To preserve the sign of the original `expression`, the bits on the left are filled in with 0 if the most significant bit (the bit farthest to the left) of `expression1` is 0, and filled in with 1 if the most significant bit is 1. Shifting a value right by one position is the equivalent of dividing by 2 and discarding the remainder.

Player

Flash 5 or later.

Example

The following example converts 65535 to a 32-bit integer, and shifts it eight bits to the right:

```
x = 65535 >> 8
```

The result of the above operation is as follows:

```
x = 255
```

This is because 65535 decimal equals 1111111111111111 binary (*sixteen 1's*), 1111111111111111 binary shifted right by eight bits is 11111111 binary, and 11111111 binary is 255 decimal. The most significant bit is 0 because the integers are 32-bit, so the fill bit is 0.

The following example converts -1 to a 32-bit integer and shifts it one bit to the right:

```
x = -1 >> 1
```

The result of the above operation is as follows:

```
x = -1
```

This is because -1 decimal equals 11111111111111111111111111111111 binary (thirty-two 1's), shifting right by one bit causes the least significant (bit farthest to the right) to be discarded and the most significant bit to be filled in with 1. The result is 11111111111111111111111111111111 (*thirty-two 1's*) binary, which represents the 32-bit integer -1.

See also

`>>= (bitwise right shift and assignment)`

>>= (bitwise right shift and assignment)

Syntax

`expression1 =>>expression2`

Arguments

`expression1` A number, string, or expression to be shifted left.

`expression2` A number, string, or expression that converts to an integer from 0 to 31.

Description

Operator (compound assignment); this operator performs a bitwise right shift operation and stores the contents as a result in *`expression1`*.

Player

Flash 5 or later.

Example

The following two expressions are equivalent:

```
A >>= B
A = (A >> B)
```

The following commented code uses the bitwise operator `>>=`. It is also an example of using all bitwise operators.

```
function convertToBinary(number)
{
var result = "";
for (var i=0; i<32; i++) {
// Extract least significant bit using bitwise AND
var lsb = number & 1;
// Add this bit to our result string
result = (lsb ? "1" : "0") + result;
// Shift number right by one bit, to see next bit
}number >>= 1;
return result;
}
convertToBinary(479)
//Returns the string
00000000000000000000000111011111
//The above string is the binary representation of the decimal
number 479.
```

See also

`<< (bitwise left shift)`

>>> (bitwise unsigned right shift)

Syntax

`expression1` `>>>` *`expression2`*

Arguments

`expression1` A number, string, or expression to be shifted right.

`expression2` A number, string, or expression that converts to an integer from 0 to 31.

Description

Operator (bitwise); the same as the bitwise right shift operator (`>>`) except that it does not preserve the sign of the original *`expression`* because the bits on the left are always filled with 0.

Player

Flash 5 or later.

Example

The following example converts -1 to a 32-bit integer and shifts it one bit to the right:

```
x = -1 >>> 1
```

The result of the above operation is as follows:

```
x = 2147483647
```

This is because -1 decimal is 11111111111111111111111111111111 binary (*thirty-two 1's*), and when you shift right (unsigned) by one bit, the least significant (rightmost) bit is discarded, and the most significant (leftmost) bit is filled with a 0. The result is:

01111111111111111111111111111111 binary,

which represents the 32-bit integer `2147483647`.

See also

`>>= (bitwise right shift and assignment)`

>>>= (bitwise unsigned right shift and assignment)

Syntax

expression1 `>>>=` *expression2*

Arguments

expression1 A number, string, or expression to be shifted left.

expression2 A number, string, or expression that converts to an integer from 0 to 31.

Description

Operator (compound assignment); performs a unsigned bitwise right shift operation and stores the contents as a result in *expression1*.

Player

Flash 5 or later.

Example

The following two expressions are equivalent:

```
A >>>= B
A = (A >>> B)
```

See also

`>>> (bitwise unsigned right shift)`
`>>= (bitwise right shift and assignment)`

add

Syntax

string1 `add` *string2*

Arguments

string1,2 Any string.

Description

Operator; concatenates two or more strings. The `add` operator replaces the Flash 4 `&` operator; Flash 4 files using the `&` operator are automatically converted to use the `add` operator for string concatenation when brought into the Flash 5 authoring environment. However, the `add` operator is deprecated in Flash 5, and use of the + operator is recommended when creating content for the Flash 5 Player. Use the `add` operator to concatentate strings if you are creating content for Flash 4 or earlier versions of the Player.

Player

Flash 4 or later.

See also

`+ (addition)`

_alpha

Syntax

```
instancename._alpha
instancename._alpha = value;
```

Arguments

instancename The name of a movie clip instance.

value A number from 0 to 100 specifying the alpha transparency.

Description

Property; sets or retrieves the alpha transparency (*value*) of the movie clip. Valid values are 0 (fully transparent) to 100 (fully opaque). Objects in a movie clip with `_alpha` set to 0 are active, even though they are invisible. For example, a button in a movie clip with `_alpha` property set to 0 can still be clicked.

Player

Flash 4 or later.

Example

The following statements set the `_alpha` property of a movie clip named `star` to 30% when the button is clicked:

```
on(release) {
   setProperty(star._alpha = 30);
}
```

or

```
on(release) {
   star._alpha = 30;
}
```

and

Syntax

condition1 and *condition2*

Arguments

condition1, *condition2* Conditions or expressions that evaluate to true or false.

Description

Operator; performs a logical AND operation in the Flash 4 Player. If both expressions evaluate to true, then the entire expression is true.

Player

Flash 4 or later. This operator has been deprecated in Flash 5, and users are encouraged to make use of the new && operator.

See also

&& (short-circuit AND)

Array (object)

The Array object allows you to access and manipulate arrays. An array is an object whose properties are identified by a number representing their position in the array. This number is sometimes referred to as the index. All arrays are zero based, which means that the first element in the array is [0], the second element is [1], and so on. In the following example, `myArray` contains the months of the year, identified by number.

```
myArray[0] = "January"
myArray[1] = "February"
myArray[2] = "March"
myArray[3] = "April"
```

To create an Array object, use the constructor `new Array`. To access the elements of an array use, the array access operator [ ].

Method summary for the Array object

Method	Description
`concat`	Concatenates the arguments and returns them as a new array.
`join`	Joins all elements of an array into a string.
`pop`	Removes the last element of an array, and returns its value.
`push`	Adds one or more elements to the end of an array and returns the array's new length.
`reverse`	Reverses the direction of an array.
`shift`	Removes the first element from an array, and returns its value.
`slice`	Extracts a section of an array and returns it as a new array.
`sort`	Sorts an array in place.
`splice`	Adds and/or removes elements from an array.
`toString`	Returns a string value representing the elements in the Array object.
`unshift`	Adds one or more elements to the beginning of an array and returns the array's new length.

Property summary for the Array object

Property	Description
`length`	Returns the length of the array.

Constructor for the Array object

Syntax

```
new Array();
new Array(length);
new Array(element0, element1, element2,...elementN);
```

Arguments

`length` An integer specifying the number of elements in the array. In the case of noncontiguous elements, the length specifies the index number of the last element in the array plus 1. For more information, see the property `Array.length`.

`element0...elementN` A list of two or more arbitrary values. The values can be numbers, names, or other elements specified in an array. The first element in an array always has the index or position 0.

Description

Constructor; allows you to access and manipulate elements in an array. Arrays are zero based and the elements are indexed by their ordinal number.

If you don't specify any arguments, a zero-length array is created.

Player

Flash 5 or later.

Example

The following example creates a new Array object with an initial length of 0:

```
myArray = new Array();
```

The following example creates the new Array object `A-Team`, with an initial length of 4:

```
A-Team = new Array("Jody", "Mary", "Marcelle", "Judy");
```

The initial elements of the `A-Team` array are as follows:

```
myArray[0] = "Jody"

myArray[1] = "Mary"

myArray[2] = "Marcelle"

myArray[3] = "Judy"
```

See also

`Array.length`

Array.concat

Syntax

myArray.concat(*value0,value1,...valueN*);

Arguments

value0,...valueN Numbers, elements, or strings to be concatenated in a new array.

Description

Method; concatenates the elements specified in the arguments, if any, and creates and returns a new array. If the arguments specify an array, the elements of that array are concatenated, rather than the array itself.

Player

Flash 5 or later.

Example

The following code concatenates two arrays:

```
alpha = new Array("a","b","c");
numeric = new Array(1,2,3);
alphaNumeric=alpha.concat(numeric);
// creates array ["a","b","c",1,2,3]
```

The following code concatenates three arrays:

```
num1=[1,3,5];
num2=[2,4,6];
num3=[7,8,9];
nums=num1.concat(num2,num3) // creates array [1,3,5,2,4,6,7,8,9]
```

Array.join

Syntax

```
myArray.join();
myArray.join(separator);
```

Arguments

`separator` A character or string that separates array elements in the returned string. If you omit this argument, a comma is used as the default separator.

Description

Method; converts the elements in an array to strings, concatenates them, inserts the specified separator between the elements, and returns the resulting string.

Player

Flash 5 or later.

Example

The following example creates an array, with three elements. It then joins the array three times: using the default separator, then a comma and a space, and then a plus sign.

```
a = new Array("Earth","Moon","Sun")
// assigns "Earth,Moon,Sun" to myVar1
myVar1=a.join();
// assigns "Earth, Moon, Sun" to myVar2
myVar2=a.join(", ");
// assigns "Earth + Moon + Sun" to myVar3
myVar3=a.join(" + ");
```

Array.length

Syntax

myArray.length;

Arguments

None.

Description

Property; contains the length of the array. This property is automatically updated when new elements are added to the array. During assignment `myArray[index] = value`; if `index` is a number, and `index+1` is a greater than the `length` property, the `length` property is updated to `index + 1`.

Player

Flash 5 or later.

Example

The following code explains how the length property is updated:

```
//initial length is 0
myArray = new Array();
//myArray.length is updated to 1
myArray[0] = 'a';
//myArray.length is updated to 2
myArray[1] = 'b';
//myArray.length is updated to 10
myArray[9] = 'c';
```

Array.pop

Syntax

myArray.pop();

Arguments

None.

Description

Method; removes the last element from an array and returns the value of that element.

Player

Flash 5 or later.

Example

The following code creates the `myPets` array containing four elements, then removes its last element:

```
myPets = ["cat", "dog", "bird", "fish"];
popped = myPets.pop();
```

Array.push

Syntax

myArray.push(*value,...*);

Arguments

value One or more values to append to the array.

Description

Method; adds one or more elements to the end of an array and returns the array's new length.

Player

Flash 5 or later.

Example

The following code creates the `myPets` array containing two elements, then adds two elements to it. After the code executes, `pushed` contains 4.

```
myPets = ["cat", "dog"];
pushed = myPets.push("bird", "fish");
```

Array.reverse

Syntax

myArray.reverse();

Arguments

None.

Description

Method; reverses the array in place.

Player

Flash 5 or later.

Example

The following is an example of using the `Array.reverse` method:

```
var numbers = [1, 2, 3, 4, 5, 6];
trace(numbers.join());
   numbers.reverse();
   trace(numbers.join());
```

Output:

```
1,2,3,4,5,6
6,5,4,3,2,1
```

Array.shift

Syntax

myArray`.shift();`

Arguments

None.

Description

Method; removes the first element from an array and returns that element.

Player

Flash 5 or later.

Example

The following code creates the array `myPets` and then removes the first element from the array:

```
myPets = ["cat", "dog", "bird", "fish"];
shifted = myPets.shift();
```

The return value is `cat`.

See also

`Array.pop`

`Array.unshift`

Array.slice

Syntax

myArray`.slice(`*start*`,` *end*`);`

Arguments

start A number specifying the index of the starting point for the slice. If *start* is a negative number, the starting point begins at the end of the array, where -1 is the last element.

end A number specifying the index of the ending point for the slice. If you omit this argument, the slice includes all elements from the start to the end of the array. If *end* is a negative number, the ending point is specified from the end of the array, where -1 is the last element.

Description

Method; extracts a slice or a substring of the array and returns it as a new array without modifying the original array. The returned array includes the *start* element and all elements up to, but not including, the *end* element.

Player

Flash 5 or later.

Array.sort

Syntax

```
myArray.sort();
myArray.sort(orderfunc);
```

Arguments

orderfunc An optional comparison function used to determine the sorting order. Given the arguments A and B, the specified ordering function should perform a sort as follows:

- -1 if A appears before B in the sorted sequence
- 0 if A = B
- 1 if A appears after B in the sorted sequence

Description

Method; sorts the array in place, without making a copy. If you omit the *orderfunc* argument, Flash sorts the elements in place using the < comparison operator.

Player

Flash 5 or later.

Example

The following example uses `Array.sort` without specifying the *orderfunc* argument:

```
var fruits = ["oranges", "apples", "strawberries",
                "pineapples", "cherries"];
   trace(fruits.join());
   fruits.sort();
   trace(fruits.join());
```

Output:

```
oranges,apples,strawberries,pineapples,cherries
apples,cherries,oranges,pineapples,strawberries
```

The following example uses `array.sort` with a specified order function:

```
var passwords = [
   "gary:foo",
   "mike:bar",
   "john:snafu",
   "steve:yuck",
   "daniel:1234"
 ];
 function order (a, b) {
   // Entries to be sorted are in form
   // name:password
   // Sort using only the name part of the
   // entry as a key.
   var name1 = a.split(':')[0];
   var name2 = b.split(':')[0];
   if (name1 < name2) {
     return -1;
   } else if (name1 > name2) {
     return 1;
   } else {
     return 0;
   }
 }
 for (var i=0; i< password.length; i++) {
   trace (passwords.join());
 }
 passwords.sort(order);
 trace ("Sorted:");
 for (var i=0; i< password.length; i++) {
   trace (passwords.join());
 }
```

Output:

```
daniel:1234
gary:foo
john:snafu
mike:bar
steve:yuck
```

Array.splice

Syntax

`myArray.splice(start, deleteCount, value0,value1...valueN);`

Arguments

start The index of the element in the array where the insertion and/or deletion begins.

deleteCount The number of elements to be deleted. This number includes the element specified in the *start* argument. If no value is specified for *deleteCount*, the method deletes all of the values from the *start* element to the last element in the array.

value Zero or more values to insert into the array at the insertion point specified in the *start* argument. This argument is optional.

Description

Method; adds and/or removes elements from an array. This method modifies the array itself without making a copy.

Player

Flash 5 or later.

Array.toString

Syntax

myArray.toString();

Arguments

None.

Description

Method; returns a string value representing the elements in the specified Array object. Every element in the array, starting with index 0 and ending with index *myArray*.length-1, is converted to a concatenated string separated by commas.

Player

Flash 5 or later.

Example

The following example creates `myArray` and converts it to a string:

```
myArray = new Array();
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
myArray[3] = 4;
myArray[4] = 5;

trace(myArray.toString())
```

Output:

```
1,2,3,4,5
```

Array.unshift

Syntax

myArray.unshift(*value1*,*value2*,...*valueN*);

Arguments

value1,...valueN One or more numbers, elements, or variables to be inserted at the beginning of the array.

Description

Method; adds one or more elements to the beginning of an array and returns the array's new length.

Player

Flash 5 or later.

Boolean (function)

Syntax

`Boolean(expression);`

Arguments

`expression` The variable, number, or string to be converted to a Boolean.

Description

Function; converts the specified argument to a Boolean, and returns the Boolean value.

Player

Flash 5 or later.

Boolean (object)

The Boolean object is a simple wrapper object with the same functionality as the standard JavaScript Boolean object. Use the Boolean object to retrieve the primitive data type or string representation of Boolean object.

Method summary for the Boolean object

Method	Description
`toString`	Returns the string representation `(true)` or `(false)` of the Boolean object.
`valueOf`	Returns the primitive value type of the specified Boolean object.

Constructor for the Boolean object

Syntax

```
new Boolean();
new Boolean(x);
```

Arguments

x A number, string, Boolean, object, movie clip, or other expression. This argument is optional.

Description

Constructor; creates an instance of the Boolean object. If you omit the *x* argument, the Boolean object is initialized with a value of `false`. If you specify *x*, the method evaluates the argument and returns the result as a Boolean value according to the following casting rules:

- If *x* is a number, the function returns `true` if *x* does not equal 0, or `false` if *x* is any other number.
- If *x* is a Boolean, the function returns *x*.
- If *x* is an object or movie clip, the function returns `true` if *x* does not equal `null`; otherwise, the function returns `false`.
- If *x* is a string, the function returns `true` if `Number(x)` does not equal 0; otherwise, the function returns `false`.

Note: To maintain compatibility with Flash 4, the handling of strings by the Boolean object is not ECMA-262 standard.

Player

Flash 5 or later.

Boolean.toString

Syntax

```
Boolean.toString();
```

Arguments

None.

Description

Method; returns the string representation, `true` or `false` of the Boolean object.

Player

Flash 5 or later.

Boolean.valueOf

Syntax

```
Boolean.valueOf();
```

Arguments

None.

Description

Method; returns the primitive value type of the specified Boolean object, and converts the Boolean wrapper object to this primitive value type.

Player

Flash 5 or later.

break

Syntax

`break;`

Arguments

None.

Description

Action; appears within a loop (`for`, `for..in`, `do...while` or `while`). The `break` action instructs Flash to skip the rest of the loop body, stop the looping action, and execute the statement following the loop statement. Use the `break` action to break out of a series of nested loops.

Player

Flash 4 or later.

Example

The following example uses the `break` action to exit an otherwise infinite loop:

```
i = 0;
while (true) {
   if (i >= 100) {
      break;
   }
   i++;
}
```

call

Syntax

`call(`*`frame`*`);`

Arguments

`frame` The name or number of the frame to call into the context of the script.

Description

Action; switches the context from the current script to the script attached to the frame being called. Local variables will not exist once the script is finished executing.

Player

Flash 4 or later. This action is deprecated in Flash 5, and it is recommended that you use the `function` action.

See also

`function`

chr

Syntax

`chr(number);`

Arguments

number The ASCII code number to convert to a character.

Description

String function; converts ASCII code numbers to characters.

Player

Flash 4 or later. This function has been deprecated in Flash 5; use of the `String.fromCharCode` method is recommended.

Example

The following example converts the number 65 to the letter “A”:

```
chr(65) = "A"
```

See also

`String.fromCharCode`

Color (object)

The Color object allows you to set and retrieve the RGB color value and color transform of movie clips. The Color object is supported by Flash 5 and later versions of the Flash Player.

You must use the constructor `new Color()` to create an instance of the Color object before calling the methods of the Color object.

Method summary for the Color object

Method	Description
`getRGB`	Returns the numeric RGB value set by the last `setRGB` call.
`getTransform`	Returns the transform information set by the last `setTransform` call.
`setRGB`	Sets the hexadecimal representation of the RGB value for a Color object.
`setTransform`	Sets the color transform for a Color object.

Constructor for the Color object

Syntax

```
new Color(target);
```

Arguments

`target` The name of the movie clip the new color is applied to.

Description

Constructor; creates a Color object for the movie clip specified by the *`target`* argument.

Player

Flash 5 or later.

Example

The following example creates a new Color object called `myColor` for the movie `myMovie`:

```
myColor = new Color(myMovie);
```

Color.getRGB

Syntax

myColor.getRGB();

Arguments

None.

Description

Method; returns the numeric values set by the last `setRGB` call.

Player

Flash 5 or later.

Example

The following code retrieves the RGB value as a hexadecimal string:

```
value = (getRGB()).toString(16);
```

See also

`Color.setRGB`

Color.getTransform

Syntax

myColor.getTransform();

Arguments

None.

Description

Method; returns the transform value set by the last `setTransform` call.

Player

Flash 5 or later.

See also

`Color.setTransform`

Color.setRGB

Syntax

myColor.setRGB(*0xRRGGBB*);

Arguments

0xRRGGBB The hexadecimal or RGB color to be set. *RR*, *GG*, and *BB* each consist of two hexadecimal digits specifying the offset of each color component.

Description

Method; specifies an RGB color for the Color object. Calling this method overrides any previous settings by the `setTransform` method.

Player

Flash 5 or later.

Example

The following example sets the RGB color value for the movie clip `myMovie`:

```
myColor = new Color(myMovie);
myColor.setRGB(0x993366);
```

See also

`Color.setTransform`

Color.setTransform

Syntax

myColor.setTransform(*colorTransformObject*);

Arguments

colorTransformObject An object created using the constructor of the generic Object object, specifying color transform values for parameters. The color transform object must have the parameters `ra`, `rb`, `ga`, `gb`, `ba`, `bb`, `aa`, `ab`, which are explained below.

Description

Method; sets color transform information for a Color object. The *colorTransformObject* argument is an object that you create using the generic Object object with parameters specifying the percentage and offset values for the red, green, blue, and alpha (transparency) components of a color, entered in a *0xRRGGBBAA* format.

The parameters for a color transformobject are defined as follows:

- `ra` is the percentage for the red component (-100 to 100).
- `rb` is the offset for the red component (-255 to 255).
- `ga` is the percentage for the green component (-100 to 100).
- `gb` is the offset for the green component (-255 to 255).
- `ba` is the percentage for the blue component (-100 to 100).
- `bb` is the offset for the blue component (-255 to 255).
- `aa` is the percentage for alpha (-100 to 100).
- `ab` is the offset for alpha (-255 to 255).

You create a color transformobject as follows:

```
myColorTransform = new Object();
myColorTransform.ra = 50;
myColorTransform.rb = 244;
myColorTransform.ga = 40;
myColorTransform.gb = 112;
myColorTransform.ba = 12;
myColorTransform.bb = 90;
myColorTransform.aa = 40;
myColorTransform.ab = 70;
```

You could also use the following syntax:

```
myColorTransform = { ra: ‘50’, rb: ‘244’, ga: ‘40’, gb: ‘112’, ba:
‘12’, bb: ‘90’, aa: ‘40’, ab: ‘70’}
```

Player

Flash 5 or later.

Example

The following example shows the process of creating a new Color object for a target movie, creating a color transformobject with the parameters defined above using the Object constructor, and passing the color transform object to a Color object using the `setTransform` method.

```
//Create a color object called myColor for the target myMovie
myColor = new Color(myMovie);
//Create a color transform object called myColorTransfrom using
//the generic Object object
myColorTransform = new Object;
// Set the values for myColorTransform
myColorTransform = { ra: '50', rb: '244', ga: '40', gb: '112', ba:
'12', bb: '90', aa: '40', ab: '70'}
//Associate the color transform object with the Color object
created for myMovie
myColor.setTransform(myColorTransform);
```

continue

Syntax

```
continue;
```

Arguments

None.

Description

Action; appears within several types of loop statements.

In a `while` loop, `continue` causes Flash to skip the rest of the loop body and jump to the top of the loop, where the condition is tested.

In a `do...while` loop, `continue` causes Flash to skip the rest of the loop body and jump to the bottom of the loop, where the condition is tested.

In a `for` loop, `continue` causes Flash to skip the rest of the loop body and jump to the evaluation of the `for` loop's post-expression.

In a `for...in` loop, `continue` causes Flash to skip the rest of the loop body and jump back to the top of the loop, where the next value in the enumeration is processed.

Player

Flash 4 or later.

See also

`do...while`
`for`
`for..in`
`while`

_currentframe

Syntax

instancename._currentframe

Arguments

instancename The name of a movie clip instance.

Description

Property (read-only); returns the number of the frame where the playhead is currently located in the Timeline.

Player

Flash 4 or later.

Example

The following example uses `_currentframe` to direct a movie to go five frames ahead of the frame containing the action:

```
gotoAndStop(_currentframe + 5);
```

Date (object)

The Date object allows you to retrieve date and time values relative to universal time (Greenwich Mean Time, now called Universal Coordinated Time) or relative to the operating system on which the Flash Player is running. To call the methods of the Date object, you must first create an instance of the Date object using the constructor.

The Date object requires the Flash 5 Player.

The methods of the Date object are not static, but apply only to the individual instance of the Date object specified when the method is called.

Method summary for Date object

Method	Description
`getDate`	Returns the day of the month of the specified Date object according to local time.
`getDay`	Returns the day of the month for the specified Date object according to local time.
`getFullYear`	Returns the four-digit year of the specified Date object according to local time.
`getHours`	Returns the hour of the specified Date object according to local time.

Method	Description
getMilliseconds	Returns the milliseconds of the specified Date object according to local time.
getMinutes	Returns the minutes of the specified Date object according to local time.
getMonth	Returns the month of the specified Date object according to local time.
getSeconds	Returns the seconds of the specified Date object according to local time.
getTime	Returns the number of milliseconds since midnight January 1, 1970, universal time, for the specified Date object.
getTimezoneOffset	Returns the difference, in minutes, between the computer's local time and the universal time.
getUTCDate	Returns the day (date) of the month of the specified Date object according to universal time.
getUTCDay	Returns the day of the week of the specified Date object according to universal time.
getUTCFullYear	Returns the four-digit year of the specified Date object according to universal time.
getUTCHours	Returns the hour of the specified Date object according to universal time.
getUTCMilliseconds	Returns the milliseconds of the specified Date object according to universal time.
getUTCMinutes	Returns the minute of the specified Date object according to universal time.
getUTCMonth	Returns the month of the specified Date object according to universal time.
getUTCSeconds	Returns the seconds of the specified Date object according to universal time.
getYear	Returns the year of the specified Date object according to local time.
setDate	Returns the day of the month of a specified Date object according to local time.
setFullYear	Sets the full year for a Date object according to local time.
setHours	Sets the hours for a Date object according to local time.
setMilliseconds	Sets the milliseconds for a Date object according to local time.
setMinutes	Sets the minutes for a Date object according to local time.

Method	Description
`setMonth`	Sets the month for a Date object according to local time.
`setSeconds`	Sets the seconds for a Date object according to local time.
`setTime`	Sets the date for the specified Date object in milliseconds.
`setUTCDate`	Sets the date of the specified Date object according to universal time.
`setUTCFullYear`	Sets the year of the specified Date object according to universal time.
`setUTCHours`	Sets the hour of the specified Date object according to universal time.
`setUTCMilliseconds`	Sets the milliseconds of the specified Date object according to universal time.
`setUTCMinutes`	Sets the minute of the specified Date object according to universal time.
`setUTCMonth`	Sets the month represented by the specified Date object according to universal time.
`setUTCSeconds`	Sets the seconds of the specified Date object according to universal time.
`setYear`	Sets the year for the specified Date object according to local time.
`toString`	Returns a string value representing the date and time stored in the specified Date object.
`Date.UTC`	Returns the number of milliseconds between midnight on January 1, 1970, universal time, and the specified time.

Constructor for the Date object

Syntax

```
new Date();
new Date(year [, month [, date [, hour [, minute [, second [,
millisecond ]]]]]] );
```

Arguments

`year` A value of 0 to 99 indicates 1900 though 1999, otherwise all 4 digits of the year must be specified.

`month` An integer from 0 (January) to 11 (December). This argument is optional.

`date` An integer from 1 to 31. This argument is optional.

`hour` An integer from 0 (midnight) to 23 (11 p.m.).

`minute` An integer from 0 to 59. This argument is optional.

`second` An integer from 0 to 59. This argument is optional.

`millisecond` An integer from 0 to 999. This argument is optional.

Description

Object; constructs a new Date object holding the current date and time.

Player

Flash 5 or later.

Example

The following example retrieves the current date and time:

```
now = new Date();
```

The following example creates a new Date object for a Gary's birthday, August 7, 1974:

```
gary_birthday = new Date (74, 7, 7);
```

The following example creates a new Date object, concatenates the returned values of the Date object methods `getMonth`, `getDate`, and `getFullYear`, and displays them in the text field specified by the variable `dateTextField`.

```
myDate = new Date();
dateTextField = (mydate.getMonth() + "/" + myDate.getDate() + "/"
+ mydate.getFullYear());
```

Date.getDate

Syntax

```
myDate.getDate();
```

Arguments

None.

Description

Method; returns the day of the month (an integer from 1 to 31) of the specified Date object according to local time.

Player

Flash 5 or later.

Date.getDay

Syntax

```
myDate.getDay();
```

Arguments

None.

Description

Method; returns the day of the month (0 for Sunday, 1 for Monday, and so on) of the specified Date object according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.getFullYear

Syntax

myDate.getFullYear();

Arguments

None.

Description

Method; returns the full year (a four-digit number, for example, 2000) of the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Example

The following example uses the constructor to create a new Date object and send the value returned by the `getFullYear` method to the Output window:

```
myDate = new Date();
trace(myDate.getFullYear());
```

Date.getHours

Syntax

myDate.getHours();

Arguments

None.

Description

Method; returns the hour (an integer from 0 to 23) of the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.getMilliseconds

Syntax

```
myDate.getMilliseconds();
```

Arguments

None.

Description

Method; returns the milliseconds (an integer from 0 to 999) of the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.getMinutes

Syntax

```
myDate.getMinutes();
```

Arguments

None.

Description

Method; returns the minutes (an integer from 0 to 59) of the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.getMonth

Syntax

```
myDate.getMonth();
```

Arguments

None.

Description

Method; returns the month (0 for January, 1 for February, and so on) of the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.getSeconds

Syntax

```
myDate.getSeconds();
```

Arguments

None.

Description

Method; returns the seconds (an integer from 0 to 59) of the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.getTime

Syntax

```
myDate.getTime();
```

Arguments

None.

Description

Method; returns the number of milliseconds (an integer from 0 to 999) since midnight January 1, 1970, universal time, for the specified Date object. Use this method to represent a specific instant in time when comparing two or more times defined in different time zones.

Player

Flash 5 or later.

Date.getTimezoneOffset

Syntax

```
mydate.getTimezoneOffset();
```

Arguments

None.

Description

Method; returns the difference, in minutes, between the computer's local time and the universal time.

Player

Flash 5 or later.

Example

The following example returns the difference between the local daylight-saving time for San Francisco and the universal time. Daylight-savings time is factored into the returned result only if the date defined in the Date object is during the daylight-savings time.

```
new Date().getTimezoneOffset();
```

The result is as follows:

```
420 (7 hours * 60 minutes/hour = 420 minutes)
```

Date.getUTCDate

Syntax

```
myDate.getUTCDate();
```

Arguments

None.

Description

Method; returns the day (date) of the month in the specified Date object, according to universal time.

Player

Flash 5 or later.

Date.getUTCDay

Syntax

```
myDate.getUTCDate();
```

Arguments

None.

Description

Method; returns the day of the week of the specified Date object, according to universal time.

Date.getUTCFullYear

Syntax

```
myDate.getUTCFullYear();
```

Arguments

None.

Description

Method; returns the four-digit year of the specified Date object, according to universal time.

Player

Flash 5 or later.

Date.getUTCHours

Syntax

```
myDate.getUTCHours();
```

Arguments

None.

Description

Method; returns the hours of the specified Date object, according to universal time.

Player

Flash 5 or later.

Date.getUTCMilliseconds

Syntax

`myDate.getUTCMilliseconds();`

Arguments

None.

Description

Method; returns the milliseconds of the specified Date object, according to universal time.

Player

Flash 5 or later.

Date.getUTCMinutes

Syntax

`myDate.getUTCMinutes();`

Arguments

None.

Description

Method; returns the minutes of the specified Date object, according to universal time.

Player

Flash 5 or later.

Date.getUTCMonth

Syntax

`myDate.getUTCMonth();`

Arguments

None.

Description

Method; returns the month of the specified Date object, according to universal time.

Player

Flash 5 or later.

Date.getUTCSeconds

Syntax

```
myDate.getUTCSeconds();
```

Arguments

None.

Description

Method; returns the seconds in the specified Date object, according to universal time.

Player

Flash 5 or later.

Date.getYear

Syntax

```
myDate.getYear();
```

Arguments

None.

Description

Method; returns the year of the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running. The year is the full year minus 1900. For example, the year 2000 is represented as 100.

Player

Flash 5 or later.

Date.setDate

Syntax

```
myDate.setDate(date);
```

Arguments

date An integer from 1 to 31.

Description

Method; sets the day of the month for the specified Date object, according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.setFullYear

Syntax

`myDate.setFullYear(year [, month [, date]] );`

Arguments

`year` A four-digit number specifying a year. Two-digit numbers do not represent years; for example, 99 is not the year 1999, but the year 99.

`month` An integer from 0 (January) to 11 (December). This argument is optional.

`date` A number from 1 to 31. This argument is optional.

Description

Method; sets the year of the specified Date object, according to local time. If the *`month`* and *`date`* arguments are specified, they are also set to local time. Local time is determined by the operating system on which the Flash Player is running.

The results of `getUTCDay` and `getDay` may change as a result of calling this method.

Player

Flash 5 or later.

Date.setHours

Syntax

`myDate.setHours(hour);`

Arguments

`hour` An integer from 0 (midnight) to 23 (11 p.m.).

Description

Method; sets the hours for the specified Date object according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.setMilliseconds

Syntax

`myDate.setMilliseconds(millisecond);`

Arguments

millisecond An integer from 0 to 999.

Description

Method; sets the milliseconds for the specified Date object according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.setMinutes

Syntax

`myDate.setMinutes(minute);`

Arguments

minute An integer from 0 to 59.

Description

Method; sets the minutes for a specified Date object according to local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.setMonth

Syntax

`myDate.setMonth(month [, date ]);`

Arguments

month An integer from 0 (January) to 11 (December).

date An integer from 1 to 31. This argument is optional.

Description

Method; sets the month for the specified Date object in local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.setSeconds

Syntax

`myDate.setSeconds(second);`

Arguments

second An integer from 0 to 59.

Description

Method; sets the seconds for the specified Date object in local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.setTime

Syntax

`myDate.setTime(millisecond);`

Arguments

millisecond An integer from 0 to 999.

Description

Method; sets the Date for the specified Date object in milliseconds.

Player

Flash 5 or later.

Date.setUTCDate

Syntax

`myDate.setUTCDate(date);`

Arguments

date An integer from 1 to 31.

Description

Method; sets the date for the specified Date object in universal time. Calling this method does not modify the other fields of the specified Date, but the `getUTCDay` and `getDay` methods may report a new value if the day of the week changes as a result of calling this method.

Player

Flash 5 or later.

Date.setUTCFullYear

Syntax

`myDate.setUTCFullYear(year [, month [, date]]);`

Arguments

year The year specified as a full four-digit year, for example, 2000.

month An integer from 0 (January) to 11 (December). This argument is optional.

date An integer from 1 to 31. This argument is optional.

Description

Method; sets the year or the specified Date object (*mydate*) in universal time.

Optionally, this method can also set the month and date represented by the specified Date object. No other fields of the Date object are modified. Calling `setUTCFullyear` may cause `getUTCDay` and `getDay` to report a new value if the day of the week changes as a result of this operation.

Player

Flash 5 or later.

Date.setUTCHours

Syntax

`myDate.setUTCHours(hour [, minute [, second [, millisecond]]]));`

Arguments

hour An integer from 0 (midnight) to 23 (11p.m.).

minute An integer from 0 to 59. This argument is optional.

second An integer from 0 to 59. This argument is optional.

millisecond An integer from 0 to 999. This argument is optional.

Description

Method; sets the hour for the specified Date object in universal time.

Player

Flash 5 or later.

Date.setUTCMilliseconds

Syntax

```
myDate.setUTCMilliseconds(millisecond);
```

Arguments

millisecond An integer from 0 to 999.

Description

Method; sets the milliseconds for the specified Date object in universal time.

Player

Flash 5 or later.

Date.setUTCMinutes

Syntax

```
myDate.setUTCMinutes(minute [, second [, millisecond]]));
```

Arguments

minute An integer from 0 to 59.

second An integer from 0 to 59. This argument is optional.

millisecond An integer from 0 to 999. This argument is optional.

Description

Method; sets the minute for the specified Date object in universal time.

Player

Flash 5 or later.

Date.setUTCMonth

Syntax

`myDate.setUTCMonth(month [, date]);`

Arguments

`month` An integer from 0 (January) to 11 (December).

`date` An integer from 1 to 31. This argument is optional.

Description

Method; sets the month, and optionally the day (date), for the specified Date object in universal time. Calling this method does not modify the other fields of the specified Date object, but the `getUTCDay` and `getDay` methods may report a new value if the day of the week changes as a result of specifying the *`date`* argument when calling `setUTCMonth`.

Player

Flash 5 or later.

Date.setUTCSeconds

Syntax

`myDate.setUTCSeconds(second [, millisecond]));`

Arguments

`second` An integer from 0 to 59.

`millisecond` An integer from 0 to 999. This argument is optional.

Description

Method; sets the seconds for the specified Date object in universal time.

Player

Flash 5 or later.

Date.setYear

Syntax

```
myDate.setYear(year);
```

Arguments

`year` A four-digit number, for example, 2000.

Description

Method; sets the year for the specified date object in local time. Local time is determined by the operating system on which the Flash Player is running.

Player

Flash 5 or later.

Date.toString

Syntax

```
myDate.toString();
```

Arguments

None.

Description

Method; returns a string value for the specified date object in a readable format.

Player

Flash 5 or later.

Example

The following example returns the information in the `dateOfBirth` Date object as a string:

```
var dateOfBirth = newDate(74, 7, 7, 18, 15);
trace (dateOfBirth.toString());
```

Output (for Pacific Standard Time):

```
Wed Aug 7 18:15:00 GMT-0700 1974
```

Date.UTC

Syntax

```
Date.UTC(year, month [, date [, hour [, minute [, second [,
millisecond ]]]]]);
```

Arguments

year A four-digit number, for example, 2000.

month An integer from 0 (January) to 11 (December).

date An integer from 1 to 31. This argument is optional.

hour An integer from 0 (midnight) to 23 (11 p.m.).

minute An integer from 0 to 59. This argument is optional.

second An integer from 0 to 59. This argument is optional.

millisecond An integer from 0 to 999. This argument is optional.

Description

Method; returns the number of milliseconds between midnight on January 1, 1970, universal time, and the time specified in the arguments. This is a static method that is invoked through the Date object constructor, not through a specific Date object. This method allows you to create a Date object that assumes universal time, whereas the Date constructor assumes local time.

Player

Flash 5 or later.

Example

The following example creates a new Date object `gary_birthday` defined in universal time. This is the universal time variation of the example used for the constructor method `new Date()`:

```
gary_birthday = new Date(Date.UTC(1974, 7, 8));
```

delete

Syntax

```
delete (reference);
```

Arguments

reference The name of variable or object to eliminate.

Description

Operator; destroys the object or variable specified as the *reference*, and returns `true` if the object was successfully deleted; otherwise returns `false`. This operator is useful for freeing up memory used by scripts, although, `delete` is an operator, it is typically used as a statement:

```
delete x;
```

The delete operator may fail and return false if the *reference* does not exist, or may not be deleted. Predefined objects and properties, and variables declared with `var`, may not be deleted.

Player

Flash 5 or later.

Example

The following example creates an object, uses it, and then deletes it once it is no longer needed:

```
account = new Object();
   account.name = 'Jon';
   account.balance = 10000;
   ...
   delete account;
```

The following example deletes a property of an object:

```
// create the new object "account"
account = new Object();
// assign property name to the account
   account.name = 'Jon';
// delete the property
delete account.name;
```

The following is another example of deleting an object property:

```
// create an Array object with length 0
array = new Array();
// Array.length is now 1
   array[0] = "abc";
// add another element to the array,Array.length is now 2
   array[1] = "def";
// add another element to array,Array.length is now 3
   array[2] = "ghi";
// array[2] is deleted, but Array.length is not changed,
   delete array[2];
```

The following example illustrates the behavior of `delete` on object references:

```
// create a new object, and assign the variable ref1 to refer to
the object
ref1 = new Object();
ref1.name = "Jody";
// copy the reference variable into a new variable, and delete
ref1
ref2 = ref1;
delete ref1;
```

If `ref1` had not been copied into `ref2`, the object would have been deleted when we deleted `ref1`, because there would be no references to it. If we were to delete `ref2`, there would no longer be any references to the object, and it would be destroyed and the memory it was using would be made available.

See also

`var`

do...while

Syntax

```
do {
statement;
} while (condition);
```

Arguments

`condition` The condition to evaluate.

`statement` The statement to execute as long as *`condition`* evaluates to `true`.

Description

Action; executes the statements, and then evaluates the condition in a loop for as long as the condition is true.

Player

Flash 4 or later.

See also

`break`
`continue`

_droptarget

Syntax

draggableInstanceName._droptarget

Arguments

draggableInstanceName The name of a movie clip instance that was the target of a `startDrag` action.

Description

Property (read-only); returns the absolute path in slash syntax notation of the movie clip instance on which the *draggableInstanceName* was dropped. The `_droptarget` property always returns a path that starts with `/`. To compare the `_droptarget` property of an instance to a reference, use `eval` to convert the returned value from slash syntax to a reference.

Player

Flash 4 or later.

Example

The following example evaluates the `_droptarget` property of the `garbage` movie clip instance and uses `eval` to convert it from slash syntax to a dot syntax reference. The `garbage` reference is then compared to the reference to the `trash` movie clip instance. If the two references are equivalent, the visibility of `garbage` is set to `false`. If they are not equivalent, the `garbage` instance is reset to its original position.

```
if (eval(garbage._droptarget) == _root.trash) {
   garbage._visible = false;
} else {
   garbage._x = x_pos;
   garbage._y = y_pos;
}
```

The variables `x_pos` and `y_pos` are set on frame 1 of the movie with the following script:

```
x_pos = garbage._x;
y_pos = garbage._y;
```

See also

`startDrag`

duplicateMovieClip

Syntax

`duplicateMovieClip(`*`target`*`, `*`newname`*`, `*`depth`*`);`

Arguments

`target` The target path of the movie to duplicate.

`newname` A unique identifier for the duplicate movie clip.

`depth` The depth level of the movie clip. The depth level is the stacking order that determines how movie clips and other objects appear when they overlap. The first movie clip that your create, or instance that you drag onto the Stage, is assigned a depth of level 0. You must assign each successive or duplicated movie clip a different depth level to prevent it from replacing movies on occupied levels or the original movie clip.

Description

Action; creates an instance of a movie clip while the movie is playing. Duplicate movie clips always start at frame 1, no matter what frame the original movie clip was on. Variables in the parent movie clip are not copied into the duplicate movie clip. If the parent movie clip is deleted the duplicate movie clip is also deleted. Use the `removeMovieClip` action or method to delete a movie clip instance created with `duplicateMovieClip`.

Player

Flash 4 or later.

Example

This statement duplicates the movie clip instance `flower` ten times. The variable `i` is used to create a new instance name and a depth.

```
on(release) {
   amount = 10;
   while(amount>0) {
      duplicateMovieClip (_root.flower, "mc" + i, i);
      setProperty("mc" + i, _x, random(275));
      setProperty("mc" + i, _y, random(275));
      setProperty("mc" + i, _alpha, random(275));
      setProperty("mc" + i, _xscale, random(50));
      setProperty("mc" + i, _yscale, random(50));
      i = i + 1;
      amount = amount-1;
   }
}
```

See also

`removeMovieClip`
`MovieClip.removeMovieClip`

else

Syntax

`else {statement(s)};`

Arguments

`statement(s)` An alternative series of statements to run if the condition specified in the `if` statement is `false`.

Description

Action; specifies the actions, clauses, arguments, or other conditional to run if the initial `if` statement returns `false`.

Player

Flash 4 or later.

See also

`if`

eq (equal–string specific)

Syntax

`expression1` `eq` *`expression2`*

Arguments

`expression1, expression2` Numbers, strings, or variables.

Description

Comparison operator; compares two expressions for equality and returns `true` if *`expression1`* is equal to *`expression2`*; otherwise, returns `false`.

Player

Flash 1 or later. This operator has been deprecated in Flash 5; use of the new == (equality) operator is recommended.

See also

`== (equality)`

escape

Syntax

```
escape(expression);
```

Arguments

expression The expression to convert into a string and encode in a URL-encoded format.

Description

Function; converts the argument to a string and encodes it in a URL-encoded format, where all alphanumeric characters are escaped with % hexadecimal sequences.

Player

Flash 5 or later.

Example

```
escape("Hello{[World]}");
```

The result of the above code is as follows:

```
Hello%7B%5BWorld%5D%7D
```

See also

`unescape`

eval

Syntax

`eval(expression);`

Arguments

`expression` A string containing the name of a variable, property, object or movie clip to retrieve.

Description

Function; accesses variables, properties, objects, or movie clip by name. If the `expression` is a variable or a property, the value of the variable or property is returned. If the `expression` is an object or movie clip, a reference to the object or movie clip is returned. If the element named in the `expression` can not be found, undefined is returned.

In Flash 4, the `eval` function was used to simulate an arrays. In Flash 5 it is recommended that you use the Array object to create arrays.

Note: The ActionScript `eval` action is not the same as the JavaScript `eval` function, and cannot be used to evaluate statements.

Player

Flash 5 or later for full functionality. You can use `eval` when exporting to the Flash 4 Player, but you must use slash notation, and can only access variables, not properties or objects.

Example

The following example uses `eval` to determine the value of the variable `x`, and sets it to the value of `y`:

```
x = 3;
y = eval("x");
```

The following example uses `eval` to reference the movie clip object associated with a movie clip instance on the Stage, `Ball`:

```
eval("_root.Ball");
```

See also

`Array (object)`

evaluate

Syntax

statement;

Arguments

None.

Description

Action; creates a new empty line and inserts a `;` for entering unique scripting statements using Expression field in the Actions panel. The `evaluate` statement also allows users who are scripting in the Flash 5 Actions panel's Normal Mode to call functions.

Player

Flash 5 or later.

_focusrect

Syntax

`_focusrect = Boolean;`

Arguments

Boolean `true` or `false`.

Description

Property (global); specifies whether a yellow rectangle appears around the button that has the current focus. The default value `true` (nonzero) displays a yellow rectangle around the currently focused button or text field as the user presses the Tab key to navigate. Specify `false` to display only the button "over" state (if any is defined) as users navigate.

Player

Flash 4 or later.

for

Syntax

```
for(init; condition; next); {
statement;
}
```

Arguments

init An expression to evaluate before beginning the looping sequence, typically an assignment expression. A `var` statement is also permitted for this argument.

condition An expression that evaluates to `true` or `false`. The condition is evaluated before each loop iteration; the loop exits when the condition evaluates to `false`.

next An expression to evaluate after each loop iteration; usually an assignment expression using the `++` (increment) or `--` (decrement) operators.

statement A statement within the body of the loop to execute.

Description

Action; a loop construct that evaluates the `init` (initialize) expression once, and then begins a looping sequence by which, as long as the *condition* evaluates to `true`, *statement* is executed and the next expression is evaluated.

Some properties can not be enumerated by the `for` or `for..in` actions. For example, the built-in methods of the Array object (`Array.sort` and `Array.reverse`) are not included in the enumeration of an Array object, and movie clip properties, such as `_x` and `_y`, are not enumerated.

Player

Flash 5 or later.

Example

The following example uses `for` to add the elements in an array:

```
for(i=0; i<10; i++) {
array [i] = (i + 5)*10;
}
```

Returns the following array:

```
[50, 60, 70, 80, 90, 100, 110, 120, 130, 140]
```

The following is an example of using `for` to perform the same action repeatedly. In the code below, the `for` loop adds the numbers from 1 to 100:

```
var sum = 0;
   for (var i=1; i<=100; i++) {
      sum = sum + i;
   }
```

See also

`++ (increment)`
`-- (decrement)`
`for..in`
`var`

for..in

Syntax

```
for(variableiterant in object){
statement;
}
```

Arguments

variableiterant The name of a variable to act as the iterant, referencing each property of an object or element in an array.

object The name of an object to be iterated over.

statement A statement to execute for each iteration.

Description

Action; loops through the properties of an object or element in an array, and executes the *statement* for each property of an object.

Some properties can not be enumerated by the `for` or `for..in` actions. For example, the built-in methods of the Array object (`Array.sort` and `Array.reverse`) are not included in the enumeration of an Array object, and movie clip properties such as `_x` and `_y` are not enumerated.

The `for...in` construct iterates over properties of objects in the iterated object's prototype chain. If the child's prototype is parent, iterating over the properties of the child with `for...in`, will also iterate over the properties of parent.

Player

Flash 5 or later.

Example

The following is an example of using `for..in` to iterate over the properties of an object:

```
myObject = { name:'Tara', age:27, city:'San Francisco' };
for (name in myObject) {
   trace ("myObject." + name + " = " + myObject[name]);
}
```

The output of this example is as follows:

```
myObject.name = Tara
myObject.age = 27
myObject.city = San Francisco
```

The following is an example of using the typeof operator with `for..in` to iterate over a particular type of child:

```
for (name in myMovieClip) {
   if (typeof (myMovieClip[name]) = "movieclip") {
      trace ("I have a movie clip child named " + name);
   }
}
```

The following example enumerates the children of a movie clip and sends each to frame 2 in their respective Timelines. The `RadioButtonGroup` movie clip is a parent with several children, `_RedRadioButton_`, `_GreenRadioButton_` and `_BlueRadioButton`.

```
for (var name in RadioButtonGroup) {
   RadioButtonGroup[name].gotoAndStop(2);
}
```

_framesloaded

Syntax

`instancename``._framesloaded`

Arguments

`instancename` The name of the movie clip instance to be evaluated.

Description

Property (read-only); the number of frames that have been loaded from a streaming movie. This property is useful for determining whether the contents of a specific frame, and all the frames before it, have loaded and are available locally in a user's browser. This property is useful for monitoring the download process of large movies. For example, you might want to display a message to users indicating that the movie is loading until a specified frame in the movie has finished loading.

Player

Flash 4 or later.

Example

The following is an example of using the `_framesloaded` property to coordinate the start of the movie to the number of frames loaded:

```
if (_framesloaded >= _totalframes) {
gotoAndPlay ("Scene 1", "start");
} else {
setProperty ("_root.loader", _xscale, (_framesloaded/
_totalframes)*100);
}
```

fscommand

Syntax

```
fscommand(command, arguments);
```

Arguments

command A string passed to the host application for any use.

arguments A string passed to the host application for any use.

Description

Action; allows the Flash movie to communicate with the program hosting the Flash Player. In a Web browser, `fscommand` calls the JavaScript function `moviename_Dofscommand` in the HTML page containing the Flash movie, where `moviename` is the name of the Flash Player as assigned by the `NAME` attribute of the `EMBED` tag or the ID property of the `OBJECT` tag. If the Flash Player is assigned the name `theMovie`, the JavaScript function called is `theMovie_Dofscommand`.

Player

Flash 3 or later.

function

Syntax

```
function functionname ([argument0, argument1,...argumentN]){
statement(s)
}
function ([argument0, argument1,...argumentN]){
statement(s)
}
```

Arguments

functionname The name of the new function.

argument Zero or more strings, numbers, or objects to pass the `function`.

statements Zero or more ActionScript statements you have defined for the body of the `function`.

Description

Action; a set of statements that you define to perform a certain task. You can *declare*, or define, a function in one location and call, or invoke, it from different scripts in a movie. When you define a function, you can also specify arguments for the function. Arguments are placeholders for values on which the function will operate. You can pass a function different arguments, also called parameters, each time you call it.

Use the `return` action in a functions *`statement(s)`* to cause a function to return, or generate, a value.

Usage 1: Declares a `function` with the specified *`functionname`*, *`arguments`*, and *`statement(s)`*. When a function is called, the function declaration is invoked. Forward referencing is permitted; within the same Action list, a function may be declared after it is called. A function declaration replaces any prior declaration of the same function. You can use this syntax wherever a statement is permitted.

Usage 2: Creates an anonymous function and returns it. This syntax is used in expressions, and is particularly useful for installing methods in objects.

Player

Flash 5 or later.

Example

(Usage 1) The following example defines the function `sqr`, which accepts one argument, and returns the `square(x*x)` of the argument. Note that if the function is declared and used in the same script, the function declaration may appear after using the function.

```
y=sqr(3);
function sqr(x) {
return x*x;
}
```

(Usage 2) The following function defines a Circle object:

```
function Circle(radius) {
  this.radius = radius;
}
```

The following statement defines an anonymous function that calculates the area of a circle and attaches it to the object `Circle` as a method:

```
Circle.prototype.area = function () {return Math.PI * this.radius
* this.radius}
```

ge (greater than or equal to–string specific)

Syntax

expression1 `ge` *expression2*

Arguments

expression1, expression2   Numbers, strings, or variables.

Description

Operator (comparison); compares *expression1* to *expression2* and returns `true` if *expression1* is greater than or equal to *expression2*; otherwise, returns `false`.

Player

Flash 4 or later. This operator has been deprecated in Flash 5; use of the new `>=` operator is recommended.

See also

`>= (greater than or equal to)`

getProperty

Syntax

`getProperty(`*instancename* , *property*`);`

Arguments

instancename   The instance name of a movie clip for which the property is being retrieved.

property   A property of a movie clip, such as an *x* or *y* coordinate.

Description

Function; returns the value of the specified *property* for the movie clip instance.

Player

Flash 4 or later.

Example

The following example retrieves the horizontal axis coordinate (`_x`) for the movie clip `myMovie`:

```
getProperty(_root.myMovie_item._x);
```

getTimer

Syntax

```
getTimer();
```

Arguments

None.

Description

Function; returns the number of milliseconds that have elapsed since the movie started playing.

Player

Flash 4 or later.

getURL

Syntax

`getURL(url [, window [, variables]]);`

Arguments

`url` The URL from which to obtain the document. The URL must be in the same subdomain as the URL where the movie currently resides.

`window` An optional argument specifying the window or HTML frame that the document should be loaded into. Enter the name of a specific window or choose from the following reserved target names:

- `_self` specifies the current frame in the current window.
- `_blank` specifies a new window.
- `_parent` specifies the parent of the current frame.
- `_top` specifies the top-level frame in the current window.

`variables` An optional argument specifying a method for sending variables. If there are no variables, omit this argument; otherwise, specify whether to load variables using a `GET` or `POST` method. `GET` appends the variables to the end of the URL, and is used for small numbers of variables. `POST` sends the variables in a separate HTTP header and is used for long strings of variables.

Description

Action; loads a document from a specific URL into a window, or passes variables to another application at a defined URL. To test this action, make sure the file to be loaded is at the specified location. To use an absolute URL (for example, *`http://www.myserver.com`*), you need a network connection.

Player

Flash 2 or later. The GET and POST options are only available to Flash 4 and later versions of the Player.

Example

This example loads a new URL into a blank browser window. The `getURL` action targets the variable `incomingAd` as the *`url`* parameter so that you can change the loaded URL without having to edit the Flash movie. The `incomingAd` variable's value is passed into Flash earlier in the movie using a `loadVariables` action.

```
on(release) {
   getURL(incomingAd, "_blank");
}
```

See also

`loadVariables`
`XML.send`
`XML.sendAndLoad`
`XMLSocket.send`

getVersion

Syntax

```
getVersion();
```

Arguments

None.

Description

Function; returns a string containing Flash Player version and platform information.

This function does not work in test-movie mode, and will only return information for versions 5 or later of the Flash Player.

Example

The following is an example of a string returned by the `getVersion` function:

```
WIN 5,0,17,0
```

This indicates that the platform is Windows, and the version number of the Flash Player is major version 5, minor version 17(5.0r17).

Player

Flash 5 or later.

gotoAndPlay

Syntax

`gotoAndPlay(`*`scene, frame`*`);`

Arguments

`scene` The scene name to which the playhead is sent.

`frame` The frame number to which the playhead is sent.

Description

Action; sends the playhead to the specified frame in a scene and plays from that frame. If no scene is specified, the playhead goes to the specified frame in the current scene.

Player

Flash 2 or later.

Example

When the user clicks a button that the `gotoAndPlay` action is assigned to, the playhead is sent to frame 16 and starts to play.

```
on(release) {
   gotoAndPlay(16);
}
```

gotoAndStop

Syntax

`gotoAndStop(`*`scene, frame`*`);`

Arguments

`scene` The scene name to which the playhead is sent.

`frame` The frame number to which the playhead is sent.

Description

Action; sends the playhead to the specified frame in a scene and stops it. If no scene is specified, the playhead is sent to the frame in the current scene.

Player

Flash 2 or later.

Example

When the user clicks a button that the `gotoAndStop` action is assigned to, the playhead is sent to frame 5 and the movie stops playing.

```
on(release) {
   gotoAndStop(5);
}
```

gt (greater than –string specific)

Syntax

`expression1` `gt` *`expression2`*

Arguments

`expression1, expression2` Numbers, strings, or variables.

Description

Operator (comparison); compares *`expression1`* to *`expression2`* and returns `true` if *`expression1`* is greater than *`expression2`*; otherwise, returns `false`.

Player

Flash 4 or later. This operator has been deprecated in Flash 5; use of the new > operator is recommended.

See also

`> (greater than)`

_height

Syntax

```
instancename._height
instancename._height = value;
```

Arguments

instancename An instance name of a movie clip for which the `_height` property is to be set or retrieved.

value An integer specifying the height of the movie in pixels.

Description

Property; sets and retrieves the height of the space occupied by a movie's content. In previous versions of Flash, `_height` and `_width` were read-only properties; in Flash 5 these properties can be set.

Player

Flash 4 or later.

Example

The following code example sets the height and width of a movie clip when the user clicks the mouse:

```
onClipEvent(mouseDown) {
   _width=200;
   _height=200;
}
```

_highquality

Syntax

```
_highquality = value;
```

Arguments

value The level of anti-aliasing applied to the movie. Specify 2 (BEST) to apply high quality with bitmap smoothing always on. Specify 1 (high quality) to apply anti-aliasing; this will smooth bitmaps if the movie does not contain animation. Specify 0 (low quality) to prevent anti-aliasing.

Description

Property (global); specifies the level of anti-aliasing applied to the current movie.

Player

Flash 4 or later.

See also

`_quality`
`toggleHighQuality`

if

Syntax

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

Arguments

conditional An expression that evaluates to `true` or `false`. For example, `if(name == "Erica")`, evaluates the variable `name` to see if it is "Erica."

statements The instructions to execute if or when the condition evaluates to `true`.

Description

Action; evaluates a condition to determine the next action in a movie. If the condition is `true`, Flash runs the statements that follow. Use `if` to create branching logic in your scripts.

Player

Flash 4 or later.

See also

`else`
`for`
`for..in`

ifFrameLoaded

Syntax

```
ifFrameLoaded(scene, frame) {
statement;}
```

```
ifFrameLoaded(frame) {
statement;}
```

Arguments

scene The scene that is being queried.

frame The frame number or frame label to load before the next statement is executed.

Description

Action; checks whether the contents of a specific frame are available locally. Use `ifFrameLoaded` to start playing a simple animation while the rest of the movie downloads to the local computer. The difference between using `_framesloaded` and `ifFrameLoaded` is that `_framesloaded` allows you to add `if`, or `else` statements, while `ifFrameLoaded` action allows you to specify a number of frames in one simple statement.

Player

Flash 3 or later. The `ifFrameLoaded` action is deprecated in Flash 5; use of the `_framesloaded` action is encouraged.

See also

`_framesloaded`

#include

Syntax

```
#include "filename.as";
```

Arguments

filename.as The filename to include; `.as` is the recommended file extension.

Description

Action; includes the contents of the file specified in the argument when the movie is tested, published, or exported. The `#include` action is invoked when you test, publish, or export. The `#include` action is checked when a syntax check occurs.

Player

N/A

Infinity

Syntax

`Infinity`

Arguments

None.

Description

Top-level variable; a predefined variable with the ECMA-262 value for infinity.

Player

Flash 5 or later.

int

Syntax

`int(`*`value`*`);`

Arguments

`value` A number to be rounded to an integer.

Description

Function; converts a decimal number to the closest integer value.

Player

Flash 4 or later. This function has been deprecated in Flash 5; use of the `Math.floor` method is recommended.

See also

`Math.floor`

isFinite

Syntax

`isFinite(expression);`

Arguments

expression The Boolean, variable, or other expression to be evaluated.

Description

Top-level function; evaluates the argument and returns `true` if it is a finite number, and `false` if it is infinity or negative infinity. The presence of infinity or negative infinity indicates a mathematical error condition such as division by 0.

Player

Flash 5 or later.

Example

The following are examples of return values for `isFinite`:

`isFinite(56)` returns `true`

`isFinite(Number.POSITIVE_INFINITY)` returns `false`

`isNaN(Number.POSITIVE_INFINITY)` returns `false`

isNaN

Syntax

`isNaN(expression);`

Arguments

expression The Boolean, variable, or other expression to be evaluated.

Description

Top-level function; evaluates the argument and returns `true` if the value is not a number (`NaN`), indicating the presence of mathematical errors.

Player

Flash 5 or later.

Example

The following illustrates the return value for `isNan`:

`isNan("Tree")` returns `true`

`isNan(56)` returns `false`

`isNaN(Number.POSITIVE_INFINITY)` returns `false`

Key (object)

The Key object is a top-level object that you can access without using a constructor. Use the methods for the Key object to build an interface that can be controlled by a user with a standard keyboard. The properties of the Key object are constants representing the keys most commonly used to control games. See Appendix B, "Keyboard Keys and Key Code Values," for a complete list of key code values.

Example

```
onClipEvent (enterFrame) {
   if(Key.isDown(Key.RIGHT)) {
      setProperty ("", _x, _x+10);
   }
}
or
onClipEvent (enterFrame) {
   if(Key.isDown(39)) {
      setProperty("", _x, _x+10);
   }
}
```

Method summary for the Key object

Method	Description
`getAscii;`	Returns the ASCII value of the last key pressed.
`getCode;`	Returns the virtual key code of the last key pressed.
`isDown;`	Returns `true` if the key specified in the argument is pressed.
`isToggled;`	Returns `true` if the Num Lock or Caps Lock key is activated.

Property summary for the Key object

All of the properties for the Key object are constants.

Property	Description
BACKSPACE	Constant associated with the key code value for the Backspace key (9).
CAPSLOCK	Constant associated with the key code value for the Caps Lock key (20).
CONTROL	Constant associated with the key code value for the Control key (17).
DELETEKEY	Constant associated with the key code value for the Delete key (46).
DOWN	Constant associated with the key code value for the Down Arrow key (40).
END	Constant associated with the key code value for the End key (35).
ENTER	Constant associated with the key code value for the Enter key (13).
ESCAPE	Constant associated with the key code value for the Escape key (27).
HOME	Constant associated with the key code value for the Home key (36).
INSERT	Constant associated with the key code value for the Insert key (45).
LEFT	Constant associated with the key code value for the Left Arrow key (37).
PGDN	Constant associated with the key code value for the Page Down key (34).
PGUP	Constant associated with the key code value for the Page Up key (33).
RIGHT	Constant associated with the key code value for the Right Arrow key (39).
SHIFT	Constant associated with the key code value for the Shift key (16).
SPACE	Constant associated with the key code value for the Spacebar (32).
TAB	Constant associated with the key code value for the Tab key (9).
UP	Constant associated with the key code value for the Up Arrow key (38).

Key.BACKSPACE

Syntax

```
Key.BACKSPACE
```

Arguments

None.

Description

Property; constant associated with the key code value for the Backspace key (9).

Player

Flash 5 or later.

Key.CAPSLOCK

Syntax

```
Key.CAPSLOCK
```

Arguments

None.

Description

Property; constant associated with the key code value for the Caps Lock key (20).

Player

Flash 5 or later.

Key.CONTROL

Syntax

```
Key.CONTROL
```

Arguments

None.

Description

Property; constant associated with the key code value for the Control key (17).

Player

Flash 5 or later.

Key.DELETEKEY

Syntax

`Key.DELETEKEY`

Arguments

None.

Description

Property; constant associated with the key code value for the Delete key (46).

Player

Flash 5 or later.

Key.DOWN

Syntax

`Key.DOWN`

Arguments

None.

Description

Property; constant associated with the key code value for the Down Arrow key (40).

Player

Flash 5 or later.

Key.END

Syntax

`Key.END`

Arguments

None.

Description

Property; constant associated with the key code value for the End key (35).

Player

Flash 5 or later.

Key.ENTER

Syntax

```
Key.ENTER
```

Arguments

None.

Description

Property; constant associated with the key code value for the Enter key (13).

Player

Flash 5 or later.

Key.ESCAPE

Syntax

```
Key.ESCAPE
```

Arguments

None.

Description

Property; constant associated with the key code value for the Escape key (27).

Player

Flash 5 or later.

Key.getAscii

Syntax

```
Key.getAscii();
```

Arguments

None.

Description

Method; returns the ASCII code of the last key pressed or released.

Player

Flash 5 or later.

Key.getCode

Syntax

```
Key.getCode();
```

Arguments

None.

Description

Method; returns the key code value of the last key pressed. Use the information in Appendix B, "Keyboard Keys and Key Code Values," to match the returned key code value with the virtual key on a standard keyboard.

Player

Flash 5 or later.

Key.HOME

Syntax

```
Key.HOME
```

Arguments

None.

Description

Property; constant associated with the key code value for the Home key (36).

Player

Flash 5 or later.

Key.INSERT

Syntax

```
Key.INSERT
```

Arguments

None.

Description

Property; constant associated with the key code value for the Insert key (45).

Player

Flash 5 or later.

Key.isDown

Syntax

`Key.isDown(`*`keycode`*`);`

Arguments

`keycode` The key code value assigned to a specific key, or a Key object property associated with a specific key. Appendix B, "Keyboard Keys and Key Code Values," lists all of the key codes associated with the keys on a standard keyboard.

Description

Method; returns `true` if the key specified in *`keycode`* is pressed. On the Macintosh, the key code values for the Caps Lock and Num Lock keys are identical.

Player

Flash 5 or later.

Key.isToggled

Syntax

`Key.isToggled(`*`keycode`*`)`

Arguments

`keycode` The key code for Caps Lock (20) or Num Lock (144).

Description

Method; returns `true` if the Caps Lock or Num Lock key is activated (toggled). On the Macintosh, the key code values for these keys are identical.

Player

Flash 5 or later.

Key.LEFT

Syntax

`Key.LEFT`

Arguments

None.

Description

Property; constant associated with the key code value for the Left Arrow key (37).

Player

Flash 5 or later.

Key.PGDN

Syntax

`Key.PGDN`

Arguments

None.

Description

Property; constant associated with the key code value for the Page Down key (34).

Player

Flash 5 or later.

Key.PGUP

Syntax

`Key.PGUP`

Arguments

None.

Description

Property; constant associated with the key code value for the Page Up key (33).

Player

Flash 5 or later.

Key.RIGHT

Syntax

`Key.RIGHT`

Arguments

None.

Description

Property; constant associated with the key code value for the Right Arrow key (39).

Player

Flash 5 or later.

Key.SHIFT

Syntax

`Key.SHIFT`

Arguments

None.

Description

Property; constant associated with the key code value for the Shift key (16).

Player

Flash 5 or later.

Key.SPACE

Syntax

`Key.SPACE`

Arguments

None.

Description

Property; constant associated with the key code value for the Spacebar (32).

Player

Flash 5 or later.

Key.TAB

Syntax

`Key.TAB`

Arguments

None.

Description

Property; constant associated with the key code value for the Tab key (9).

Player

Flash 5 or later.

Key.UP

Syntax

`Key.UP`

Arguments

None.

Description

Property; constant associated with the key code value for the Up Arrow key (38).

Player

Flash 5 or later.

le (less than or equal to – string specific)

Syntax

`expression1` `le` *`expression2`*

Arguments

`expression1``,`*`expression2`* Numbers, strings, or variables.

Description

Operator (comparison); compares *`expression1`* to *`expression2`* and returns `true` if *`expression1`* is less than or equal to *`expression2`*; otherwise, returns `false`.

Player

Flash 4 or later. This operator has been deprecated in Flash 5; use of the new `<=` operator is recommended.

See also

`<= (less than or equal to)`

length

Syntax

```
length(expression);
length(variable);
```

Arguments

`expression` Any string.

`variable` The name of a variable.

Description

String function; returns the length of the specified string or variable name.

Player

Flash 4 or later. This function, along with all of the string functions, has been deprecated in Flash 5. It is recommended that you use the methods and `length` property of the String object to perform the same operations.

Example

The following example returns the value of the string `Hello`:

```
length("Hello");
```

The result is 5.

See also

`" " (string delimiter)`
`String.length`

_level

Syntax

```
_levelN;
```

Arguments

N A nonnegative integer specifying a depth level. By default, `_level` is set to 0, the movie at the base of the hierarchy.

Description

Property; a reference to the root movie Timeline of *levelN*. You must load movies using the `loadMovie` action, before targeting them using the `_level` property.

In the Flash Player, movies are assigned a number according to the order in which they were loaded. The movie that was loaded first is loaded at the bottom level, level 0. The movie in level 0 sets the frame rate, background color, and frame size for all subsequently loaded movies. Movies are then stacked in higher numbered levels above the movie in level 0. The level where a movie clip resides is also referred to as the depth level or depth.

Player

Flash 4 or later.

Example

The following example stops the Timeline of the movie in level 0:

```
_level0.stop();
```

The following example sends the Timeline of the movie in level 4 to frame 5. The movie in level 4 must have previously been loaded with a `loadMovie` action:

```
_level4.gotoAndStop(5);
```

See also

`loadMovie`
`MovieClip.swapDepths`

loadMovie

Syntax

`loadMovie(url [,location/target, variables]]);`

Arguments

`url` An absolute or relative URL for the SWF file to load. A relative path must be relative to the SWF. The URL must be in the same subdomain as the URL where the movie currently resides. For use in the Flash Player or for testing in test-movie mode in the Flash authoring environment, all SWF files must be stored in the same folder, and the file names cannot include folder or disk drive specifications.

`target` An optional argument specifying a target movie clip that will be replaced by the loaded movie. The loaded movie inherits the position, rotation, and scale properties of the targeted movie clip. Specifying a *`target`* is the same as specifying the *`location`* (level) of a target movie; you should not specify both.

`location` An optional argument specifying the level into which the movie is loaded. The loaded movie inherits the position, rotation, and scale properties of the targeted movie clip. To load the new movie in addition to existing movies, specify a level that is not occupied by another movie. To replace an existing movie with the loaded movie, specify a level that is currently occupied by another movie. To replace the original movie and unload every level, load the new movie into level 0. The movie in level 0 sets the frame rate, background color, and frame size for all other loaded movies.

`variables` An optional argument specifying a method for sending variables associated with the movie to load. The argument must be the string "GET" or "POST." If there are no variables, omit this argument; otherwise, specify whether to load variables using a `GET` or `POST` method. `GET` appends the variables to the end of the URL, and is used for small numbers of variables. `POST` sends the variables in a separate HTTP header and is used for long strings of variables.

Description

Action; plays additional movies without closing the Flash Player. Normally, the Flash Player displays a single Flash Player movie (SWF file) and then closes. The `loadMovie` action lets you display several movies at once or switch between movies without loading another HTML document.

You can load movies into level that already have SWF files loaded. If you do, the new movie will replace the existing SWF file. If you load a new movie into Level 0, every level is unloaded, and Level 0 is replaced with the new file. Use the `loadVariables` action to keep the active movie, and update the variables with new values.

Use the `unloadMovie` action to remove movies loaded with the `loadMovie` action.

Player

Flash 3 or later.

Example

This `loadMovie` statement is attached to a navigation button labeled Products. There is an invisible movie clip on the Stage with the instance name `dropZone`. The loadMovie action uses this movie clip as the target parameter to load the products in the SWF file, into the correct position on the Stage:

```
on(release) {
   loadMovie("products.swf",_root.dropZone);
}
```

See also

`unloadMovie`
`_level`

loadVariables

Syntax

`loadVariables (url ,location [, variables]);`

Arguments

url An absolute or relative URL where the variables are located. The host for the URL must be in the same subdomain as the movie when accessed using a Web browser.

location A level or target to receive the variables. In the Flash Player, movie files are assigned a number according to the order in which they were loaded. The first movie loads into the bottom level (level 0). Inside the `loadMovie` action, you must specify a level number for each successive movie. This argument is optional.

variables An optional argument specifying a method for sending variables. If there are no variables, omit this argument; otherwise, specify whether to load variables using a `GET` or `POST` method. `GET` appends the variables to the end of the URL and is used for small numbers of variables. `POST` sends the variables in a separate HTTP header and is used for long strings of variables.

Description

Action; reads data from an external file, such as a text file or text generated by a CGI script, Active Server Pages (ASP), or Personal Home Page (PHP), and sets the values for variables in a movie or movie clip. This action can also be used to update variables in the active movie with new values.

The text at the specified URL must be in the standard MIME format *application/x-www-urlformencoded* (a standard format used by CGI scripts). The movie and the variables to be loaded must reside at the same subdomain. Any number of variables can be specified. For example, the phrase below defines several variables:

```
company=Macromedia&address=600+Townsend&city=San+Francisco&zip=94
103
```

Player

Flash 4 or later.

Example

This example loads information from a text file into text fields in the main Timeline (level 0). The variable names of the text fields must match the variable names in the data.txt file.

```
on(release) {
   loadVariables("data.txt", 0);
}
```

See also

`getURL`
`MovieClip.loadMovie`
`MovieClip.loadVariables`

lt (less than – string specific)

Syntax

`*expression1* lt *expression2*`

Arguments

expression1, *expression2* Numbers, strings, or variables.

Description

Operator (comparison); compares *expression1* to *expression2* and returns `true` if *expression1* is less than *expression2*; otherwise, returns `false`.

Player

Flash 4 or later. This operator has been deprecated in Flash 5; use of the new `<` (less than) operator is recommended.

See also

`< (less than)`

Math (object)

The Math object is a top-level object that you can access without using a constructor.

Use the methods and properties of this object to access and manipulate mathematical constants and functions. All of the properties and methods of the Math object are static, and must be called using the syntax `Math.method(argument)` or `Math.constant`. In ActionScript, constants are defined with the maximum precision of double-precision IEEE-754 floating-point numbers.

The Math object is fully supported in the Flash 5 Player. In the Flash 4 Player, methods of the Math object work, but they are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Several of the Math object methods take the radian of an angle as an argument. You can use the equation below to calculate radian values, or simply pass the equation (entering a value for degrees) for the radian argument.

To calculate a radian value, use this formula:

```
radian = Math.PI/180 * degree
```

The following is an example of passing the equation as an argument to calculate the sine of a 45-degree angle:

`Math.SIN(Math.PI/180 * 45)` is the same as `Math.SIN(.7854)`

Method summary for the Math object

Method	Description
`abs`	Computes an absolute value.
`acos`	Computes an arc cosine.
`asin`	Computes an arc sine.
`atan`	Computes an arc tangent.
`atan2`	Computes an angle from the x-axis to the point.
`ceil`	Rounds a number up to the nearest integer.
`cos`	Computes a cosine.
`exp`	Computes an exponential value.
`floor`	Rounds a number down to the nearest integer.
`log`	Computes a natural logarithm.
`max`	Returns the larger of the two integers.
`min`	Returns the smaller of the two integers.
`pow`	Computes *x* raised to the power of the *y*.
`random`	Returns a pseudo-random number between 0.0 and 1.0.
`round`	Rounds to the nearest integer.
`sin`	Computes a sine.
`sqrt`	Computes a square root.
`tan`	Computes a tangent.

Property summary for the Math object

All of the properties for the Math object are constants.

Property	Description
`E`	Euler's constant and the base of natural logarithms (approximately 2.718).
`LN2`	The natural logarithm of 2 (approximately 0.693).
`LOG2E`	The base 2 logarithm of *e* (approximately 1.442).
`LN10`	The natural logarithm of 10 (approximately 2.302).
`LOG10E`	The base 10 logarithm of *e* (approximately 0.434).
`PI`	The ratio of the circumference of a circle to its diameter (approximately 3.14159).
`SQRT1_2`	The reciprocal of the square root of 1/2 (approximately 0.707).
`SQRT2`	The square root of 2 (approximately 1.414).

Math.abs

Syntax

```
Math.abs(x);
```

Arguments

`x` Any number.

Description

Method; computes and returns an absolute value for the number specified by the argument `x`.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.acos

Syntax

```
Math.acos(x);
```

Arguments

x A number from -1.0 to 1.0.

Description

Method; computes and returns the arc cosine of the number specified in the argument *x*, in radians.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.asin

Syntax

```
Math.asin(x);
```

Arguments

x A number from -1.0 to 1.0.

Description

Method; computes and returns the arc sine for the number specified in the argument *x*, in radians.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.atan

Syntax

`Math.atan(x);`

Arguments

x Any number.

Description

Method; computes and returns the arc tangent for the number specified in the argument *x*. The return value is between negative pi divided by 2, and positive pi divided by 2.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.atan2

Syntax

`Math.atan2(y, x);`

Arguments

x A number specifying the *x* coordinate of the point.

y A number specifying the *y* coordinate of the point.

Description

Method; computes and returns the arc tangent of *y*/*x* in radians. The return value represents the angle opposite the opposite angle of a right triangle, where *x* is the adjacent side length and *y* is the opposite side length.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.ceil

Syntax

`Math.ceil(x);`

Arguments

x A number or expression.

Description

Method; returns the ceiling of the specified number or expression. The ceiling of a number is the closest integer that is greater than or equal to the number.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.cos

Syntax

`Math.cos(x);`

Arguments

x An angle measured in radians.

Description

Method; returns the cosine (a value from -1.0 to 1.0) of the angle specified by the argument *x*. The angle *x* must be specified in radians. Use the information outlined in the introduction to the Math object to calculate a radian.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.E

Syntax

`Math.E`

Arguments

None.

Description

Constant; a mathematical constant for the base of natural logarithms, expressed as *e*. The approximate value of *e* is 2.71828.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.exp

Syntax

`Math.exp(x);`

Arguments

`x` The exponent; a number or expression.

Description

Method; returns the value of the base of the natural logarithm (*e*), to the power of the exponent specified in the argument `x`. The constant `Math.E` can provide the value of *e*.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.floor

Syntax

```
Math.floor(x);
```

Arguments

x A number or expression.

Description

Method; returns the floor of the number or expression specified in the argument *x*. The floor is the closest integer that is less than or equal to the specified number or expression.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Example

The following returns a value of 12:

```
Math.floor(12.5);
```

Math.log

Syntax

```
Math.log(x);
```

Arguments

x A number or expression with a value greater than 0.

Description

Method; returns the natural logarithm of the argument *x*.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.LOG2E

Syntax

`Math.LOG2E`

Arguments

None.

Description

Constant; a mathematical constant for the base-2 logarithm of the constant *e* (`Math.E`), expressed as $\log_e 2$, with an approximate value of 1.442695040888963387.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.LOG10E

Syntax

`Math.LOG10E`

Arguments

None.

Description

Constant; a mathematical constant for the base-10 logarithm of the constant *e* (`Math.E`), expressed as $\log_{10} e$, with an approximate value of 0.43429448190325181667.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.LN2

Syntax

```
Math.LN2
```

Arguments

None.

Description

Constant; a mathematical constant for the natural logarithm of 2, expressed as $\log_e 2$, with an approximate value of 0.69314718055994528623.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.LN10

Syntax

```
Math.LN10
```

Arguments

None.

Description

Constant; a mathematical constant for the natural logarithm of 10, expressed as $\log_e 10$, with an approximate value of 2.3025850929940459011.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.max

Syntax

```
Math.max(x , y);
```

Arguments

x A number or expression.

y A number or expression.

Description

Method; evaluates *x* and *y* and returns the larger value.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.min

Syntax

```
Math.min(x , y);
```

Arguments

x A number or expression.

y A number or expression.

Description

Method; evaluates *x* and *y* and returns the smaller value.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.PI

Syntax

```
Math.PI
```

Arguments

None.

Description

Constant; a mathematical constant for the ratio of the circumference of a circle to its diameter, expressed as pi, with a value of 3.14159265358979.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.pow

Syntax

```
Math.pow(x , y);
```

Arguments

x A number to be raised to a power.

y A number specifying a power the argument *x* is raised to.

Description

Method; computes and returns *x* to the power of *y*, *xy*.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.random

Syntax

`Math.random();`

Arguments

None.

Description

Method; returns a pseudo-random number between 0.0 and 1.0.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

See also

`random`

Math.round

Syntax

`Math.round(`*`x`*`);`

Arguments

x Any number.

Description

Method; rounds the value of the argument *x* up or down to the nearest integer and returns the value.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.sin

Syntax

`Math.sin(x);`

Arguments

x An angle measured in radians.

Description

Method; computes and returns the sine of the specified angle in radians. Use the information outlined in the introduction to the Math object to calculate a radian.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

See also

`Math (object)`

Math.sqrt

Syntax

`Math.sqrt(x);`

Arguments

x Any number or expression greater than or equal to 0.

Description

Method; computes and returns the square root of the specified number.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.SQRT1_2

Syntax

`Math.SQRT1_2`

Arguments

None.

Description

Constant; a mathematical constant for the reciprocal of the square root of one half (1/2), with an approximate value of 0.707106781186.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.SQRT2

Syntax

`Math.SQRT2`

Arguments

None.

Description

Constant; a mathematical constant for the square root of 2, with an approximate value of 1.414213562373.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

Math.tan

Syntax

`Math.tan(x);`

Arguments

x An angle measured in radians.

Description

Method; computes and returns the tangent of the specified angle. Use the information outlined in the introduction to the Math object to calculate a radian.

Player

Flash 5 or later. In the Flash 4 Player, the methods and properties of the Math object are emulated using approximations and may not be as accurate as the non-emulated math functions supported by the Flash 5 Player.

maxscroll

Syntax

`variable_name.maxscroll = x`

Arguments

variable_name The name of a variable associated with a text field.

x The line number that is the maximum value allowed for the `scroll` property, based on the height of the text field. This is a read-only value set by Flash.

Description

Property; a read-only property that works with the `scroll` property to control the display of information in a text field. This property can be retrieved, but not modified.

Player

Flash 4 or later.

See also

`scroll`

mbchr

Syntax

`mbchr(number);`

Arguments

`number` The number to convert to a multibyte character.

Description

String function; converts an ASCII code number to a multibyte character.

Player

Flash 4 or later. This function has been deprecated in Flash 5; use of `String.fromCharCode` method is encouraged.

See also

`String.fromCharCode`

mblength

Syntax

`mblength(string);`

Arguments

`string` A string.

Description

String function; returns the length of the multibyte character string.

Player

Flash 4 or later. This function has been deprecated in Flash 5; use of the String object and methods is recommended.

mbord

Syntax

`mbord(character);`

Arguments

`character` The character to convert to a multibyte number.

Description

String function; converts the specified character to a multibyte number.

Player

Flash 4 or later. This function has been deprecated in Flash 5; use of the `String.charCodeAt` method is recommended.

See also

`String.fromCharCode`

mbsubstring

Syntax

`mbsubstring(value, index, count);`

Arguments

`value` The multibyte string from which to extract a new multibyte string.

`index` The number of the first character to extract.

`count` The number of characters to include in the extracted string, not including the index character.

Description

String function; extracts a new multibyte character string from a multibyte character string.

Player

Flash 4 or later. This function is deprecated in Flash 5; use of the `string.substr` method is recommended.

See also

`String.substr`

Mouse (object)

Use the methods of the Mouse object to hide and show the cursor in the movie. The mouse pointer is visible by default, but you can hide it and implement a custom cursor that you create using a movie clip.

Mouse method summary

Method	Description
`hide`	Hides the cursor in the movie.
`show`	Displays the cursor in the movie.

Mouse.hide

Syntax

`Mouse.hide();`

Arguments

None.

Description

Method; hides the cursor in a movie. The cursor is visible by default.

Player

Flash 5 or later.

Example

The following code, attached to a movie clip on the main Timeline, hides the standard cursor and sets the *x* and *y* positions of the `customCursor` movie clip instance to the *x* and *y* mouse positions in the main Timeline:

```
onClipEvent(enterFrame){
   Mouse.hide();
   customCursorMC_x = _root._xmouse;
   customCursorMC_y = _root._ymouse;
}
```

See also

`_xmouse`
`_ymouse`
`Mouse.show`

Mouse.show

Syntax

```
Mouse.show();
```

Arguments

None.

Description

Method; makes the cursor visible in a movie. The cursor is visible by default.

Player

Flash 5 or later.

See also

```
_xmouse
_ymouse
Mouse.show
```

MovieClip (object)

The methods for the MovieClip object privide the same functionality as the standard actions that target movie clips. There are also additional methods that provide functionality that is not available using the standard actions listed in the Actions category of the Actions panel. You do not need to use a constructor method in order to call the methods of the MovieClip object; instead, you reference movie clip instances by name, using the following syntax:

```
anyMovieClip.play();
anyMovieClip.gotoAndPlay(3);
```

Method summary for the MovieClip object

Method	Description
`attachMovie`	Attaches a movie in the library.
`duplicateMovieClip`	Duplicates the specified movie clip.
`getBounds`	Returns the minimum and maximum *x* and *y* coordinates of a movie in a specified coordinate space.
`getBytesLoaded`	Returns the number of bytes loaded for the specified movie clip.
`getBytesTotal`	Returns the size of the movie clip in bytes.
`getURL`	Retrieves a document from a URL.
`globalToLocal`	Converts the point object from Stage coordinates to the local coordinates of the specified movie clip.
`gotoAndPlay`	Sends the playhead to a specific frame in the movie clip and plays the movie.
`gotoAndStop`	Sends the playhead to a specific frame in the movie clip and stops the movie.
`hitTest`	Returns `true` if bounding box of the specified movie clip intersects the bounding box of the target movie clip.
`loadMovie`	Loads the specified movie into the movie clip.
`loadVariables`	Loads variables from a URL or other location into the movie clip.
`localToGlobal`	Converts a Point object from the local coordinates of the movie clip to the global Stage coordinates.
`nextFrame`	Sends the playhead to the next frame of the movie clip.
`play`	Plays the specified movie clip.
`prevFrame`	Sends the playhead to the previous frame of the movie clip.
`removeMovieclip`	Removes the movie clip from the Timeline if it was created with a duplicateMovieClip action or method or the attachMovie method.
`startDrag`	Specifies a movie clip as draggable and begins dragging the movie clip.
`stop`	Stops the currently playing movie.
`stopDrag`	Stops the dragging of any movie clip that is being dragged.
`swapDepths`	Swaps the depth level of specified movie with the movie at a specific depth level.
`unloadMovie`	Removes a movie loaded with `loadMovie`.

MovieClip.attachMovie

Syntax

```
anyMovieClip.attachMovie(idName, newname, depth);
```

Arguments

idName The name of the movie in the library to attach. This is the name entered in the Identifier field in the Symbol Linkage Properties dialog box.

newname A unique instance name for the movie clip being attached.

depth An integer specifying the depth level where the movie is placed.

Description

Method; creates a new instance of a movie in the library and attaches it to the movie specified by *anyMovieClip*. Use the `removeMovieClip` or `unloadMovie` action or method to remove a movie attached with `attachMovie`.

Player

Flash 5 or later.

See also

`removeMovieClip`
`unloadMovie`
`MovieClip.removeMovieClip`
`MovieClip.unloadMovie`

MovieClip.duplicateMovieClip

Syntax

`anyMovieClip.duplicateMovieClip(newname, depth);`

Arguments

newname A unique identifier for the duplicate movie clip.

depth A number specifying the depth level where the movie specified is to be placed.

Description

Method; creates an instance of the specified movie clip while the movie is playing. Duplicated movie clips always start playing at frame 1, no matter what frame the original movie clip is on when the `duplicateMovieClip` method is called. Variables in the parent movie clip are not copied into the duplicate movie clip. If the parent movie clip is deleted the duplicate movie clip is also deleted. Movie clips added with `duplicateMovieClip` can be deleted with `removeMovieClip` action or method.

Player

Flash 5 or later.

See also

`removeMovieClip`
`MovieClip.removeMovieClip`

MovieClip.getBounds

Syntax

`anyMovieClip.getBounds(targetCoordinateSpace);`

Arguments

targetCoordinateSpace The target path of the Timeline whose coordinate space you want to use as a reference point.

Description

Method; returns the minimum and maximum *x* and *y* coordinate values of the MovieClip for the target coordinate space specified in the argument. The return object will contain the properties `{xMin, xMax, yMin, yMax}`. Use the `localToGlobal` and `globalToLocal` methods of the MovieClip object to convert the movie clip's local coordinates to Stage coordinates, or Stage coordinates to local coordinates respectively.

Player

Flash 5 or later.

Example

The following example uses `getBounds` to retrieve the bounding box of the `myMovieClip` instance in the coordinate space of the main movie:

```
myMovieClip.getBounds(._root);
```

See also

`MovieClip.globalToLocal`
`MovieClip.localToGlobal`

MovieClip.getBytesLoaded

Syntax

`anyMovieClip.getBytesLoaded();`

Arguments

None.

Description

Method; returns the number of bytes loaded (streamed) for the specified Movie Clip object. Because internal movie clips load automatically, the return result for this method and `MovieClip.getBytesTotal` will be the same if the specified Movie Clip object references an internal movie clip. This method is intended for use on loaded movies. You can compare the value of `getBytesLoaded` with the value of `getBytesTotal` to determine what percentage of an external movie has loaded.

Player

Flash 5 or later.

MovieClip.getBytesTotal

Syntax

`anyMovieClip.getBytesTotal();`

Arguments

None.

Description

Method; returns the size, in bytes, of the specified Movie Clip object. For movie clips that are external, (the root movie or a movie clip that is being loaded into a target or a level) the return value is the size of the SWF file.

Player

Flash 5 or later.

MovieClip.getURL

Syntax

`anyMovieClip.getURL(URL [,window, variables]]);`

Arguments

URL The URL from which to obtain the document.

window An optional argument specifying the name, frame, or expression specifying the window or HTML frame that the document is loaded into. You can also use one of the following reserved target names: `_self` specifies the current frame in the current window, `_blank` specifies a new window, `_parent` specifies the parent of the current frame, `_top` specifies the top-level frame in the current window.

variables An optional argument specifying a method for sending variables associated with the movie to load. If there are no variables, omit this argument; otherwise, specify whether to load variables using a `GET` or `POST` method. `GET` appends the variables to the end of the URL, and is used for small numbers of variables. `POST` sends the variables in a separate HTTP header and is used for long strings of variables.

Description

Method; loads a document from the specified URL into the specified window. The `getURL` method can also be used to pass variables to another application defined at the URL using a `GET` or `POST` method.

Player

Flash 5 or later.

MovieClip.globalToLocal

Syntax

anyMovieClip.globalToLocal(*point*);

Arguments

point The name or identifier of an object created with the generic Object object specifying the *x* and *y* coordinates as properties.

Description

Method; converts the *point* object from Stage (global) coordinates to the movie clip's (local) coordinates.

Player

Flash 5 or later.

Example

The following example converts the global *x* and *y* coordinates of the *point* object to the local coordinates of the movie clip:

```
onClipEvent(mouseMove) {
   point = new object();
   point.x = _root._xmouse;
   point.y = _root._ymouse;
   globalToLocal(point);
   _root.out = _xmouse + " === " + _ymouse;
   _root.out2 = point.x + " === " + point.y;
   updateAfterEvent();
}
```

See also

MovieClip.localToGlobal
MovieClip.getBounds

MovieClip.gotoAndPlay

Syntax

anyMovieClip.gotoAndPlay(*frame*);

Arguments

frame The frame number to which the playhead will be sent.

Description

Method; starts playing the movie at the specified frame.

Player

Flash 5 or later.

MovieClip.gotoAndStop

Syntax

anyMovieClip.gotoAndStop(*frame*);

Arguments

frame The frame number to which the playhead will be sent.

Description

Method; stops the movie playing at the specified frame.

Player

Flash 5 or later.

MovieClip.hitTest

Syntax

```
anyMovieClip.hitTest(x, y, shapeFlag);
anyMovieClip.hitTest(target);
```

Arguments

x The *x* coordinate of the hit area on the Stage.

y The *y* coordinate of the hit area on the Stage.

The *x* and *y* coordinates are defined in the global coordinate space.

target The target path of the hit area that may intersect or overlap with the instance specified by *anyMovieClip*. The *target* usually represents a button or text-entry field.

shapeFlag A Boolean value specifying whether to evaluate the entire shape of the specified instance (*true*), or just the bounding box (*false*). This argument can only be specified if the hit area is identified using *x* and *y* coordinate arguments.

Description

Method; evaluates the instance specified by *anyMovieClip* to see if it overlaps or intersects with the hit area identified by the *target* or *x* and *y* coordinate arguments.

Usage 1compares the *x* and *y* coordinates to the shape or bounding box of the specified instance, according to the *shapeFlag* setting. If *shapeFlag* is set to `true`, only the area actually occupied by the instance on the Stage is evaluated, and if *x* and *y* overlap at any point, a value of `true` is returned. This is useful for determining if the movie clip is within a specified hit, or hotspot, area.

Usage 2 evaluates the bounding boxes of the *target* and specified instance, and returns `true` if they overlap or intersect at any point.

Player

Flash 5 or later.

Example

The following example uses `hitTest` with the `x_mouse` and `y_mouse` properties to determine whether the mouse is over the target's bounding box:

```
if (hitTest( _root._xmouse, _root._ymouse, false));
```

The following example uses `hitTest` to determine if the movie clip `ball` overlaps or intersects with the movie clip `square`:

```
if(_root.ball, hittest(_root.square)){
   trace("ball intersects square");
}
```

See also

```
MovieClip.localToGlobal
MovieClip.globalToLocal
MovieClip.getBounds
```

MovieClip.loadMovie

Syntax

anyMovieClip.loadMovie(*url* [,*variables*]);

Arguments

url An absolute or relative URL for the SWF file to load. A relative path must be relative to the SWF. The URL must be in the same subdomain as the URL where the movie currently resides. For use in the Flash Player or for testing in test-movie mode in the Flash authoring environment, all SWF files must be stored in the same folder, and the file names cannot include folder or disk drive specifications.

variables An optional argument specifying a method for sending variables associated with the movie to load. The argument must be the string "GET" or "POST." If there are no variables, omit this argument; otherwise, specify whether to load variables using a `GET` or `POST` method. `GET` appends the variables to the end of the URL and is used for small numbers of variables. `POST` sends the variables in a separate HTTP header and is used for long strings of variables.

Description

Method; plays additional movies without closing the Flash Player. Normally, the Flash Player displays a single Flash Player movie (SWF file) and then closes. The `loadMovie` method allows you display several movies at once or switch between movies without loading another HTML document.

Use the `unloadMovie` action to remove movies loaded with the `loadMovie` action.

Use the `loadVariables` method to keep the active movie, and update the variables with new values.

Player

Flash 5 or later.

See also

`MovieClip.loadVariables`
`MovieClip.unloadMovie`

MovieClip.loadVariables

Syntax

`anyMovieClip.loadVariables(url, variables);`

Arguments

`url` The absolute or relative URL for the external file. The host for the URL must be in the same subdomain as the movie clip.

`variables` The method for retrieving the variables. `GET` appends the variables to the end of the URL, and is used for small numbers of variables. `POST` sends the variables in a separate HTTP header and is used for long strings of variables.

Description

Method; reads data from an external file and sets the values for variables in a movie or movie clip. The external file can be a text file generated by a CGI script, Active Server Pages (ASP), or PHP, and can contain any number of variables.

This method can also be used to update variables in the active movie with new values.

This method requires that the text at the URL be in the standard MIME format: *`application/x-www-urlformencoded`* (CGI script format).

Player

Flash 5 or later.

See also

`MovieClip.loadMovie`

MovieClip.localToGlobal

Syntax

anyMovieClip`.localToGlobal(`*point*`);`

Arguments

point The name or identifier of an object created with the Object object, specifying the *x* and *y* coordinates as properties.

Description

Method; converts the *point* object from the movie clip's (local) coordinates, to Stage (global) coordinates.

Player

Flash 5 or later.

Example

The following example converts *x* and *y* coordinates of the *point* object, from the movie clip's coordinates (local) to the Stage coordinates (global). The local *x* and *y* coordinates are specified using `xmouse` and `ymouse` to retrieve the *x* and *y* coordinates of the mouse position.

```
onClipEvent(mouseMove) {
   point = new object();
   point.x = _xmouse;
   point.y = _ymouse;
   _root.out3 = point.x + " === " + point.y;
   _root.out = _root._xmouse + " === " + _root._ymouse;
   localToGlobal(point);
   _root.out2 = point.x + " === " + point.y;
   updateAfterEvent();
}
```

See also

`MovieClip.globalToLocal`

MovieClip.nextFrame

Syntax

anyMovieClip`.nextFrame();`

Arguments

None.

Description

Method; sends the playhead to the next frame of the movie clip.

Player

Flash 5 or later.

MovieClip.play

Syntax

anyMovieClip`.play();`

Arguments

None.

Description

Method; plays the movie clip.

Player

Flash 5 or later.

MovieClip.prevFrame

Syntax

anyMovieClip`.prevFrame();`

Arguments

None.

Description

Method; sends the playhead to the previous frame and stops it.

Player

Flash 5 or later.

MovieClip.removeMovieClip

Syntax

anyMovieClip`.removeMovieClip();`

Arguments

None.

Description

Method; removes a movie clip instance created with the `duplicateMovieclip` action, or the `duplicateMovieClip` or `attachMovie` methods of the MovieClip object.

Player

Flash 5 or later.

See also

`MovieClip.loadMovie`
`MovieClip.attachMovie`

MovieClip.startDrag

Syntax

`anyMovieClip.startDrag([lock, left, right, top, bottom]);`

Arguments

`lock` A Boolean value specifying whether the draggable movie clip is locked to the center of the mouse position (*`true`*), or locked to the point where the user first clicked on the movie clip (*`false`*). This argument is optional.

`left`, *`top`*, *`right`*, *`bottom`* Values relative to the coordinates of the movie clip's parent that specify a constraint rectangle for the movie clip. These arguments are optional.

Description

Method; allows the user to drag the specified movie clip. The movie remains draggable until explicitly stopped by calling the `stopDrag` method, or until another movie clip is made draggable. Only one movie clip is draggable at a time.

Player

Flash 5 or later.

See also

`MovieClip.stopDrag`
`_droptarget`

MovieClip.stop

Syntax

`anyMovieClip.stop();`

Arguments

None.

Description

Method; stops the movie clip currently playing.

Player

Flash 5 or later.

MovieClip.stopDrag

Syntax

`anyMovieClip.stopDrag();`

Arguments

None.

Description

Method; ends a drag action implemented with the `startDrag` method. A movie remains draggable until a `stopDrag` method is added, or until another movie becomes draggable. Only one movie clip is draggable at a time.

Player

Flash 5 or later.

See also

`_droptarget`
`MovieClip.startDrag`

MovieClip.swapDepths

Syntax

```
anyMovieClip.swapDepths(depth);
anyMovieClip.swapDepths(target);
```

Arguments

`target` The movie clip instance whose depth that is being swapped by the instance specified in *`anyMovieClip`*. Both instances must have the same parent movie clip.

`depth` A number specifying the depth level where the *`anyMovieClip`* is to be placed.

Description

Method; swaps the stacking, or *z*, order (depth level) of the specified instance with the movie specified by the *`target`* argument, or with the movie that currently occupies the *`depth`* level specified in the argument. Both movies must have the same parent movie clip. Swapping the depth level of movie clips has the effect of moving one movie in front of or behind the other. If a movie is tweening when this method is called, the tweening is stopped.

Player

Flash 5 or later.

See also

`_level`

MovieClip.unloadMovie

Syntax

```
anyMovieClip.unloadMovie();
```

Arguments

None.

Description

Method; removes a movie clip loaded with the `loadMovie` or `attachMovie` MovieClip methods.

Player

Flash 5 or later.

See also

`MovieClip.loadMovie`
`MovieClip.attachMovie`

_name

Syntax

```
instancename._name
instancename._name = value;
```

Arguments

instancename An instance name of a movie clip for which the `_name` property is to be set or retrieved.

value A string that specifies a new instance name.

Description

Property; specifies the movie clip instance name.

Player

Flash 4 or later.

NaN

Syntax

`NaN`

Arguments

None.

Description

Variable; a predefined variable with the IEEE-754 value for NaN (Not a Number).

Player

Flash 5 or later.

ne (not equal – string specific)

Syntax

`expression1 ne expression2`

Arguments

`expression1, expression2` Numbers, strings, or variables.

Description

Operator (comparison); compares *`expression1`* to *`expression2`* and returns `true` if *`expression1`* is not equal to *`expression2`*; otherwise, returns `false`.

Player

Flash 4 or later. This operator has been deprecated in Flash 5; use of the new `!=` (not equal) operator is recommended.

See also

`!= (inequality)`

new

Syntax

`new constructor();`

Arguments

constructor A function followed by any optional arguments in the parentheses. The function is usually the name of the type of object (For example, Array, Math, Number, Object) to be constructed.

Description

Operator; creates a new, initially anonymous object, calls the function identified by the *constructor* argument, passes any optional arguments in the parentheses, and passes the newly created object as a value of the keyword `this`. The constructor function can then use `this` to instantiate the new object.

The `_prototype_` property of the constructor function's object is copied into the `_proto_` property of the new object. As a result, the new object supports all of the methods and properties specified in the constructor function's Prototype object.

Player

Flash 5 or later.

Example

The following example creates the objects `book1` and `book2` using the new operator.

```
function Book(name, price)
{
   this.name = name;
   this.price = price;
}
book1 = new Book("Confederacy of Dunces", 19.95);
book2 = new Book("The Floating Opera", 10.95);
```

See also

`[] (array access operator)`
`{} (object initializer)`

The constructor method section within an object entry.

newline

Syntax

```
newline;
```

Arguments

None.

Description

Constant; inserts a carriage return character (`{`) inserting a blank line into the ActionScript code. Use `newline` to make space for information that is retrieved by a function or action in your code.

Player

Flash 4 or later.

nextFrame

Syntax

```
nextFrame();
```

Arguments

None.

Description

Action; sends the playhead to the next frame and stops it.

Player

Flash 2 or later.

Example

When the user clicks a button that a `nextFrame` action is assigned to, the playhead is sent to the next frame.

```
on (release) {
   nextFrame(5);
}
```

nextScene

Syntax

```
nextScene();
```

Arguments

None.

Description

Action; sends the playhead to frame 1 of the next scene and stops it.

Player

Flash 2 or later.

Example

This action is assigned to a button that, when pressed and released, sends the playhead to frame 1 of the next scene.

```
on(release) {
   nextScene();
}
```

not

Syntax

`not` *`expression`*

Arguments

`expression` Any variable or other expression that converts to a Boolean value.

Description

Operator; performs a logical NOT operation in the Flash 4 Player.

Player

Flash 4 or later. This operator has been deprecated in Flash 5; use of the new `!` (logical NOT) operator is recommended.

See also

`! (logical NOT)`

null

Syntax

`null`

Arguments

None.

Description

Keyword; a special value that can be assigned to variables, or returned by a function if no data was provided. You can use `null` to represent values that are missing or do not have a defined data type.

Player

Flash 5 or later.

Example

In a numeric context, `null` evaluates to 0. Equality tests can be performed with `null`. In this statement, a binary tree node has no left child, so the field for its left child could be set to `null`.

```
if (tree.left == null) {
   tree.left = new TreeNode();
}
```

Number (function)

Syntax

`Number(expression);`

Arguments

expression The string, Boolean, or other expression to convert to a number.

Description

Function; converts the argument *x* to a number and returns a value as follows:

If *x* is a number, the return value is *x*.

If *x* is a Boolean, the return value is 1 if *x* is `true`, 0 if *x* is `false`.

If *x* is a string, the function attempts to parse *x* as a decimal number with an optional trailing exponent, that is, 1.57505e-3.

If *x* is undefined, the return value is 0.

This function is used to convert Flash 4 files containing deprecated operators that are imported into the Flash 5 authoring environment. See the `&` operator for more information.

Player

Flash 4 or later.

See also

`Number (object)`

Number (object)

The Number object is a simple wrapper object for the number data type, which means that you can manipulate primitive numeric values using the methods and properties associated with the Number object. The functionality provided by this object is identical to that of the JavaScript Number object.

You must use the `Number` constructor when calling the methods of the Number object, but you do not need to use the constructor when calling the properties of the Number object. The following examples specify the syntax for calling the methods and properties of the Number object:

This is an example of calling the `toString` method of the Number object:

```
myNumber = new Number(1234);
myNumber.toString();
```

Returns a string containing the binary representation of the number 1234.

This is an example of calling the `MIN_VALUE` property (also called a constant) of the Number object:

```
smallest = Number.MIN_VALUE
```

Method summary for the Number object

Method	Description
`toString`	Returns the string representation of a Number object.
`valueOf`	Returns the primitive value of a Number object.

Property summary for the Number object

Property	Description
`MAX_VALUE`	Constant representing the largest representable number (double-precision IEEE-754). This number is approximately 1.79769313486231 58e+308.
`MIN_VALUE`	Constant representing the smallest representable number (double-precision IEEE-754). This number is approximately 5e-324.
`NaN`	Constant representing the value for Not a Number (NaN).
`NEGATIVE_INFINITY`	Constant representing the value for negative infinity.
`POSITIVE_INFINITY`	Constant representing the value for positive infinity. This value is the same as the global variable Infinity.

Constructor for the Number object

Syntax

`myNumber = new Number(`*`value`*`);`

Arguments

`value` The numeric value of the Number object being created, or a value to be converted to a number.

Description

Constructor; creates a new Number object. You must use the `Number` constructor when using the `toString` and `valueOf` methods of the Number object. You do not use a constructor when using the properties of the Number object. The `new Number` constructor is primarily used as a placeholder. An instance of the Number object is not the same as the `Number` function that converts an argument to a primitive value.

Player

Flash 5 or later.

Example

The following code constructs new Number objects:

```
n1 = new Number(3.4);
n2 = new Number(-10);
```

See also

`Number (function)`

Number.MAX_VALUE

Syntax

`Number.MAX_VALUE`

Arguments

None.

Description

Property; the largest representable number (double-precision IEEE-754). This number is approximately 1.79E+308.

Player

Flash 5 or later.

Number.MIN_VALUE

Syntax

`Number.MIN_VALUE`

Arguments

None.

Description

Property; the smallest representable number (double-precision IEEE-754). This number is approximately 5e-324.

Player

Flash 5 or later.

Number.NaN

Syntax

`Number.NaN`

Arguments

None.

Description

Property; the IEEE-754 value representing Not A Number (`NaN`).

Player

Flash 5 or later.

Number.NEGATIVE_INFINITY

Syntax

`Number.NEGATIVE_INFINITY`

Arguments

None.

Description

Property; returns the IEEE-754 value representing negative infinity. This value is the same as the global variable `Infinity`.

Negative infinity is a special numeric value that is returned when a mathematical operation or function returns a negative value larger than can be represented.

Player

Flash 5 or later.

Number.POSITIVE_INFINITY

Syntax

```
Number.POSITIVE_INFINITY
```

Arguments

None.

Description

Property; returns the IEEE-754 value representing positive infinity. This value is the same as the global variable `Infinity`.

Positive infinity is a special numeric value that is returned when a mathematical operation or function returns a value larger than can be represented.

Player

Flash 5 or later.

Number.toString

Syntax

```
myNumber.toString(radix);
```

Arguments

`radix` Specifies the numeric base (from 2 to 36) to use for the number-to-string conversion. If you do not specify the *`radix`* argument, the default value is 10.

Description

Method; returns the string representation of the specified Number object (*`myNumber`*).

Player

Flash 5 or later.

Example

The following example uses the `Number.toString` method, specifying 2 for the *`radix`* argument:

```
myNumber = new Number (1000);
(1000).toString(2);
```

Returns a string containing the binary representation of the number 1000.

Number.valueOf

Syntax

`myNumber.valueOf();`

Arguments

None.

Description

Method; returns the primitive value type of the specified Number object, and converts the Number wrapper object to the primitive value type.

Player

Flash 5 or later.

Object (object)

The generic Object object is at the root of the ActionScript class hierarchy. The functionality of the generic Object object is a small subset of that provided by the JavaScript Object object.

The generic Object object requires the Flash 5 Player.

Method summary for the Object object

Method	Description
`toString`	Converts the specified object to a string, and returns it.
`valueOf`	Returns the primitive value of an Object object.

Constructor for the Object object

Syntax

```
new Object();
new Object(value);
```

Arguments

`value` A number, Boolean, or string to be converted to an object. This argument is optional. If you do not specify `value`, the constructor creates a new object with no defined properties.

Description

Constructor; creates a new Object object.

Player

Flash 5 or later.

See also

```
Sound.setTransform
Color.setTransform
```

Object.toString

Syntax

```
myObject.toString();
```

Arguments

None.

Description

Method; converts the specified object to a string, and returns it.

Player

Flash 5 or later.

Object.valueOf

Syntax

```
myObject.valueOf();
```

Arguments

None.

Description

Method; returns the primitive value of the specified object. If the object does not have a primitive value, the object itself is returned.

Player

Flash 5 or later.

onClipEvent

Syntax

```
onClipEvent(movieEvent);{
...
}
```

Arguments

A `movieEvent` is a trigger event that executes actions that are assigned to a movie clip instance. Any of the following values can be specified for the `movieEvent` argument:

- `load` The action is initiated as soon as the movie clip is instantiated and appears in the Timeline.
- `unload` The action is initiated in the first frame after the movie clip is removed from the Timeline. The actions associated with the `Unload` movie clip event are processed before any actions are attached to the affected frame.
- `enterFrame` The action is initiated as each frame is played, similar to actions attached to a movie clip. The actions associated with the `OnEnterFrame` movie clip event are processed after any actions that are attached to the affected frames.
- `mouseMove` The action is initiated every time the mouse is moved. Use the `_xmouse` and `_ymouse` properties to determine the current mouse position.
- `mouseDown` The action is initiated when the left mouse button is pressed.
- `mouseUp` The action is initiated when the left mouse button is released.
- `keyDown` The action is initiated when a key is pressed. Use the `Key.getCode` method to retrieve information about the last key pressed.
- `keyUp` The action is initiated when a key is released. Use the `Key.getCode` method to retrieve information about the last key pressed.
- `data` The action is initiated when data is received in a `loadVariables` or `loadMovie` action. When specified with a `loadVariables` action, the `data` event occurs only once, when the last variable is loaded. When specified with a `loadMovie` action, the `data` event occurs repeatedly, as each section of data is retrieved.

Description

Handler; triggers actions defined for a specific instance of a movie clip.

Player

Flash 5 or later.

Example

The following statement includes the script from an external file when the movie clip instance is loaded and first appears on the Timeline:

```
onClipEvent(load) {
   #include "myScript.as"
}
```

The following example uses `onClipEvent` with the `keyDown` movie event. The `keyDown` movie event is usually used in conjunction with one or more methods and properties associated with the Key object. In the script below, `key.getCode` is used to find out which key the user has pressed; the returned value is associated with the `RIGHT` or `LEFT` Key object properties, and the movie is directed accordingly.

```
onClipEvent(keyDown) {
   if (Key.getCode() == Key.RIGHT) {
      }  _parent.nextFrame();
   else if (Key.getCode() == Key.LEFT){
      _parent.prevFrame();
}
```

The following example uses `onClipEvent` with the `mouseMove` movie event. The the `xmouse` and `ymouse` properties track the position of the mouse.

```
onClipEvent(mouseMove) {
stageX=_root.xmouse;
stageY=_root.ymouse;
}
```

See also

`on(mouseEvent)`
`Key (object)`
`_xmouse`
`_ymouse`

on(mouseEvent)

Syntax

```
on(mouseEvent) {
statement;
}
```

Arguments

`statement` The instructions to execute when the mouseEvent takes place.

A `mouseEvent` action can have one of the following arguments:

- `press` The mouse button is pressed while the pointer is over the button.
- `release` The mouse button is released while the pointer is over the button.
- `releaseOutside` The mouse button is released while the pointer is outside the button.
- `rollOver` The mouse pointer rolls over the button.
- `rollOut` The pointer rolls outside of the button area.
- `dragOver` While the pointer is over the button, the mouse button has been pressed while rolled outside the button, and then rolled back over the button.
- `dragOut` While the pointer is over the button, the mouse button is pressed and then rolls outside the button area.
- `keyPress ("key")` The specified *key* is pressed. The *key* portion of the argument is specified using any of the key codes listed in the Appendix B, "Keyboard Keys and Key Code Values," or any of the key constants listed in the Property summary for the Key object.

Description

Handler; specifies the mouse event, or keypress that trigger an action.

Player

Flash 2 or later.

Example

In the following script, the `startDrag` action executes when the mouse is pressed and the conditional script is executed when the mouse is released and the object is dropped:

```
on(press) {
   startDrag("rabbit");
}
on(release) {
   if(getproperty("", _droptarget) == target) {
      setProperty ("rabbit", _x, _root.rabbit_x);
      setProperty ("rabbit", _y, _root.rabbit_y);
   } else {
_root.rabbit_x = getProperty("rabbit", _x);
_root.rabbit_y = getProperty("rabbit", _y);
_root.target = "pasture";
   }
   trace(_root.rabbit_y);
   trace(_root.rabbit_x);
   stopDrag();
}
```

See also

`Key (object)`
`onClipEvent`

or

Syntax

`condition1 or condition2`

Arguments

condition1,2 An expression that evaluates to `true` or `false`.

Description

Operator; evaluates *condition1* and *condition2*, and if either expression is `true`, then the whole expression is `true`.

Player

Flash 4 or later. This operator has been deprecated in Flash 5, and users are encouraged to make use of the new `||` operator.

See also

`|| (OR)`

ord

Syntax

`ord(character);`

Arguments

character The character to convert to an ASCII code number.

Description

String function; converts characters to ASCII code numbers.

Player

Flash 4 or later. This function has been deprecated in Flash 5, and it is recommended that you use the methods and properties of the `String` object instead.

See also

`String (object)`

_parent

Syntax

```
_parent.property = x
_parent._parent.property = x
```

Arguments

`property` The property being specified for the current and parent movie clip.

`x` The value set for the property. This is an optional argument and may not need to be set, depending on the property.

Description

Property; specifies or returns a reference to the movie clip that contains the current movie clip. The current movie clip is the movie clip containing the currently executing script. Use `_parent` to specify a relative path.

Player

Flash 4 or later.

Example

In the following example the movie clip `desk` is a child of the movie clip `classroom`. When the script below executes inside the movie clip `desk`, the playhead will jump to frame 10 in the Timeline of the movie clip `classroom`.

```
_parent.gotoAndStop(10);
```

See also

`_root`
`targetPath`

parseFloat

Syntax

`parseFloat(string);`

Arguments

`string` The string to parse and convert to a floating-point number.

Description

Function; converts a string to a floating-point number. The function parses and returns the numbers in the string, until the parser reaches a character that is not a part of the initial number. If the string does not begin with a number that can be parsed, `parseFloat` returns `NaN` or 0. White space preceding valid integers is ignored, as are trailing non-numeric characters.

Player

Flash 5 or later.

Example

The following are examples of using `parseFloat` to evaluate various types of numbers:

`parseFloat("-2")` returns `-2`

`parseFloat("2.5")` returns `2.5`

`parseFloat("3.5e6")` returns `3.5e6`, or `3500000`

`parseFloat("foobar")` returns `NaN`

parseInt

Syntax

`parseInt(expression, radix);`

Arguments

`expression` The string, floating-point number, or other expression to parse and convert to a integer.

`radix` An integer representing the radix (base) of the number to parse. Legal values are from 2 and 36. This argument is optional.

Description

Function; converts a string to an integer. If the specified string in the arguments cannot be converted to a number, the function returns `NaN` or 0. Integers beginning with 0 or specifying a radix of 8 are interpreted as octal numbers. Integers beginning with 0x are interpreted as hexadecimal numbers. White space preceding valid integers is ignored, as are trailing nonnumeric characters.

Player

Flash 5 or later.

Example

The following are examples of using `parseInt` to evaluate various types of numbers:

`parseInt("3.5")` returns `3.5`

`parseInt("bar")` returns `NaN`

`parseInt("4foo")` returns `4`

Hexadecimal conversion:

`parseInt("0x3F8")` returns `1016`

`parseInt("3E8", 16)` returns `1000`

Binary conversion:

`parseInt("1010", 2)` returns `10` (the decimal representation of the binary 1010)

Octal number parsing (in this case the octal number is identified by the radix, 8):

`parseInt("777", 8)` returns `511` (the decimal representation of the octal 777)

play

Syntax

```
play();
```

Arguments

None.

Description

Action; moves the playhead forward in the Timeline.

Player

Flash 2 or later.

Example

The following code uses an `if` statement to check the value of a name the user enters. If the user enters `Steve`, the `play` action is called and the playhead moves forward in the Timeline. If the user enters anything other than `Steve`, the movie does not play and a text field with the variable name `alert` is displayed.

```
stop();
if (name = "Steve") {
   play();
} else {
   alert = "You are not Steve!";
}
```

prevFrame

Syntax

```
prevFrame();
```

Arguments

None.

Description

Action; sends the playhead to the previous frame and stops it.

Player

Flash 2 or later.

Example

When the user clicks a button that a `prevFrame` action is assigned to, the playhead is sent to the previous frame.

```
on(release) {
   prevFrame(5);
}
```

See also

`MovieClip.prevFrame`

prevScene

Syntax

```
prevScene();
```

Arguments

None.

Description

Action; sends the playhead to frame 1 of the previous scene and stops it.

Player

Flash 2 or later.

See also

`nextScene`

print

Syntax

```
print (target, "bmovie");
print (target, "bmax");
print (target, "bframe");
```

Arguments

`target` The instance name of movie clip to print. By default, all of the frames in the movie are printed. If you want to print only specific frames in the movie, designate frames for printing by attaching a `#P` frame label to those frames in the authoring environment.

`bmovie` Designates the bounding box of a specific frame in a movie as the print area for all printable frames in the movie. Attach a `#b` label (in the authoring environment) to designate the frame whose bounding box you want to use as the print area.

`bmax` Designates a composite of all of the bounding boxes, of all the printable frames, as the print area. Specify the *`bmax`* argument when the printable frames in your movie vary in size.

`bframe` Designates that the bounding box of each printable frame be used as the print area for that frame. This changes the print area for each frame and scales the objects to fit the print area. Use *`bframe`* if you have objects of different sizes in each frame and want each object to fill the printed page.

Description

Action; prints the *`target`* movie clip according to the printer modifier specified in the argument. If you want to print only specific frames in the target movie, attach a `#P` frame label to the frames you want to print. Although the `print` action results in higher quality prints than the `printAsBitmap` action, it cannot be used to print movies that use alpha transparencies or special color effects.

If you do not specify a print area argument, the print area is determined by the Stage size of the loaded movie by default. The movie does not inherit the main movie's Stage size. You can control the print area by specifying the *`bmovie`*, *`bmax`*, or *`bframe`* arguments.

All of the printable elements in a movie must be fully loaded before printing can begin.

The Flash Player printing feature supports PostScript and non-PostScript printers. Non-PostScript printers convert vectors to bitmaps.

Player

Flash 5 or later.

Example

The following example will print all of the printable frames in `myMovie` with the print area defined by the bounding box of the frame with the `#b` frame label attached:

```
print("myMovie","bmovie");
```

The following example will print all of the printable frames in `myMovie` with a print area defined by the bounding box of each frame:

```
print("myMovie","bframe");
```

See also

`printAsBitmap`

printAsBitmap

Syntax

```
printAsBitmap(target, "bmovie");
printAsBitmap(target, "bmax");
printAsBitmap(target, "bframe");
```

Arguments

`target` The instance name of movie clip to print. By default, all of the frames in the movie are printed. If you want to print only specific frames in the movie, designate frames for printing by attaching a `#P` frame label to those frames in the authoring environment.

`bmovie` Designates the bounding box of a specific frame in a movie as the print area for all printable frames in the movie. Attach a `#b` label (in the authoring environment) to designate the frame whose bounding box you want to use as the print area.

`bmax` Designates a composite of all of the bounding boxes, of all the printable frames, as the print area. Specify the `bmax` argument when the printable frames in your movie vary in size.

`bframe` Designates that the bounding box of each printable frame be used as the print area for that frame. This changes the print area for each frame and scales the objects to fit the print area. Use `bframe` if you have objects of different sizes in each frame and want each object to fill the printed page.

Description

Action; prints the `target` movie clip as a bitmap. Use `printAsBitmap` to print movies that contain frames with objects that use transparency or color effects. The `printAsBitmap` action prints at the highest available resolution of the printer in order to maintain as much definition and quality as possible. To calculate the printable file size of a frame designated to print as a bitmap, multiply pixel width by pixel height by printer resolution.

If your movie does not contain alpha transparencies or color effects, it is recommended that you use the `print` action for better quality results.

By default, the print area is determined by the Stage size of the loaded movie. The movie does not inherit the main movie's Stage size. You can control the print area by specifying the `bmovie`, `bmax`, or `bframe` arguments.

All of the printable elements in a movie must be fully loaded before printing can begin.

The Flash Player printing feature supports PostScript and non-PostScript printers. Non-PostScript printers convert vectors to bitmaps.

Player

Flash 5 or later.

See also

`print`

_quality

Syntax

```
_quality
_quality = x;
```

Arguments

`x` A string specifying one of the following values:

LOW Low rendering quality. Graphics are not antialiased, bitmaps are not smoothed.

MEDIUM Medium rendering quality. Graphics are antialiased using a 2x2 grid, but bitmaps are not smoothed. Suitable for movies that do not contain text.

HIGH High rendering quality. Graphics are antialiased using a 4x4 grid, and bitmaps are smoothed if the movie is static. This is the default rendering quality setting used by Flash.

BEST Very high rendering quality. Graphics are antialiased using a 4x4 grid, and bitmaps are always smoothed.

Description

Property (global); sets or retrieves the rendering quality used for a movie.

Player

Flash 5 or later.

Example

The following example sets the rendering for `oldQuality` to HIGH:

```
oldQualtiy = _quality
_quality = "HIGH";
```

See also

`_highquality`

random

Syntax

```
random();
```

Arguments

value The highest integer for which `random` will return a value.

Description

Function; returns a random integer between 0 and the integer specified in the *value* argument.

Player

Flash 4. This function is deprecated in Flash 5; use of the `Math.random` method is recommended.

Example

The following use of `random` returns a value of 0, 1, 2, 3, or 4:

```
random(5);
```

See also

`Math.random`

removeMovieClip

Syntax

```
removeMovieClip(target);
```

Arguments

target The target path of a movie clip instance created with `duplicateMovieClip`, or the instance name of a movie clip created with the `attachMovie` or `duplicateMovie` methods of the MovieClip object.

Description

Action; deletes a movie clip instance that was created with the `attachMovie` or `duplicateMovieClip` methods of the MovieClip object, or with the `duplicateMovieClip` action.

Player

Flash 4 or later.

See also

```
duplicateMovieClip
MovieClip.duplicateMovieClip
MovieClip.attachMovie
MovieClip.removeMovieClip
```

return

Syntax

```
return[expression];
return;
```

Arguments

expression A type, string, number, array, or object to evaluate and return as a value of the function. This argument is optional.

Description

Action; specifies the value returned by a function. When the return action is executed, the *expression* is evaluated and returned as a value of the function. The return action causes the function to stop executing. If the `return` statement is used alone, or if Flash does not encounter a `return` statement during the looping action, it returns `null`.

Player

Flash 5 or later.

Example

The following is an example of using `return`:

```
function sum(a, b, c){
return a + b + c;
}
```

See also

`function`

_root

Syntax

```
_root;
_root.movieClip;
_root.action;
```

Arguments

movieClip The instance name of a movie clip.

action The value set for the property. This is an optional argument and may not need to be set depending on the property.

Description

Property; specifies or returns a reference to the root movie Timeline. If a movie has multiple levels, the root movie Timeline is on the level containing the currently executing script. For example, if a script in level 1 evaluates `_root`, level 1 is returned.

Specifying `_root` is the same as using the slash notation (`/`) to specify an absolute path within the current level.

Player

Flash 4 or later.

Example

The following example stops the Timeline of the level containing the currently executing script:

```
_root1.stop();
```

The following example sends the Timeline in the current level to frame 3:

```
_root.gotoAndStop(3);
```

See also

`_parent`
`targetPath`

_rotation

Syntax

```
instancename._rotation
instancename._rotation = integer
```

Arguments

integer The number of degrees to rotate the movie clip.

instancename The movie clip to rotate.

Description

Property; specifies the rotation of the movie clip in degrees.

Player

Flash 4 or later.

scroll

Syntax

```
variable_name.scroll = x
```

Arguments

variable_name The name of a variable associated with a text field.

x The line number of the topmost visible line in the text field. You can specify this value or use the default value of 1. The Flash Player updates this value as the user scrolls up and down the text field.

Description

Property; controls the display of information in a text field associated with a variable. The `scroll` property defines where the text field begins displaying content; after you set it, the Flash Player updates it as the user scrolls through the text field. The `scroll` property is useful for directing users to a specific paragraph in a long passage, or creating scrolling text fields. This property can be retrieved and modified.

Player

Flash 4 or later.

See also

`maxscroll`

Selection (object)

The Selection object allows you to set and control the currently focused editable text field. The currently focused editable text field is the field where the user's mouse pointer is currently placed. Selection-span indexes are zero-based (where the first position is 0, the second position is 1, and so on).

There is no constructor method for the Selection object, as there can only be one currently focused field at a time.

Method summary for the Selection object

Method	Description
`getBeginIndex`	Returns the index at the beginning of selection span. Returns -1 if there is no index or currently selected field.
`getCaretIndex`	Returns the current caret position in the currently focused selection span. Returns -1 if there is no caret position or currently focused selection span.
`getEndIndex`	Returns the index at the end of the selection span. Returns -1 if there is no index or currently selected field.
`getFocus`	Returns the name of the variable for currently focused editable text field. Returns null if there is no currently focused editable text field.
`setFocus`	Focuses the editable text field associated with the variable specified in the argument.
`setSelection`	Sets the beginning and ending indexes of the selection span.

Selection.getBeginIndex

Syntax

```
Selection.getBeginIndex();
```

Arguments

None.

Description

Method; returns index at the beginning of the selection span. If no index exists or no field currently has the focus, the method returns -1. Selection span indexes are zero-based (where the first position is 0, the second position is 1, and so on).

Player

Flash 5 or later.

Selection.getCaretIndex

Syntax

```
Selection.getCaretIndex();
```

Arguments

None.

Description

Method; returns the index of the blinking cursor position. If there is no blinking mouse pointer displayed, the method returns -1. Selection span indexes are zero-based (where the first position is 0, the second position is 1, and so on).

Player

Flash 5 or later.

Selection.getEndIndex

Syntax

```
Selection.getEndIndex();
```

Arguments

None.

Description

Method; returns the ending index of the currently focused selection span. If no index exists, or if there is no currently focused selection span, the method returns -1. Selection span indexes are zero-based (where the first position is 0, the second position is 1, and so on).

Player

Flash 5 or later.

Selection.getFocus

Syntax

```
Selection.getFocus();
```

Arguments

None.

Description

Method; returns the name of the variable of the currently focused editable text field. If no text field is currently focused, the method returns `null`.

Player

Flash 5 or later.

Example

The following code returns the name of the variable:

```
_root.anyMovieClip.myTextField.
```

Selection.setFocus

Syntax

```
Selection.setFocus(variable);
```

Arguments

`variable` A string specifying the name of a variable associated with a text field using dot or slash notation.

Description

Method; focuses the editable text field associated with the specified *`variable`*.

Player

Flash 5 or later.

Selection.setSelection

Syntax

```
Selection.setSelection(start, end);
```

Arguments

start The beginning index of the selection span.

end The ending index of the selection span.

Description

Method; sets the selection span of the currently focused text field. The new selection span will begin at the index specified in the `start` argument, and end at the index specified in the `end` argument. Selection span indexes are zero-based (where the first position is 0, the second position is 1, and so on). This method has no effect if there is no currently focused text field.

Player

Flash 5 or later.

set

Syntax

```
variable = expression;
set(variable, expression);
```

Arguments

variable The name of the container that holds the value of the `expression` argument.

expression The value (or a phrase that can be evaluated to a value) that is assigned to the variable.

Description

Action; assigns a value to a variable. 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 movie plays, you can record and save information about what the user has done, record values that change as the movie plays, or evaluate whether a condition is `true` or `false`.

Variables can hold either numbers or strings of characters. Each movie and movie clip has its own set of variables, and each variable has its own value independent of variables in other movies or movie clips.

ActionScript is an untyped language. That means that variables do not need to be explicitly defined as containing either a number or a string. Flash interprets the data type as an integer or string accordingly.

Use the `set` statement in conjunction with the `call` action to pass or return values.

Player

Flash 4 or later.

Example

This example sets a variable called `orig_x_pos` that stores the original *x* axis position of the `ship` movie clip in order to reset the ship to its starting location later in the movie:

```
on(release) {
   set(x_pos, getProperty ("ship", _x ));
}
```

This is equivalent to writing the following:

```
on(release) {
   orig_x_pos = getProperty ("ship", _x );
}
```

See also

`var`
`call`

setProperty

Syntax

`setProperty(target, property, expression);`

Arguments

`target` The path to the instance name of the movie clip whose property is being set.

`property` The property to be set.

`expression` The value to which the property is set.

Description

Action; changes the property of a movie clip as the movie plays.

Player

Flash 4 or later.

Example

This statement sets the `_alpha` property of a movie clip named `star` to 30 percent when the button is clicked:

```
on(release) {
   setProperty("star", _alpha = 30);
}
```

See also

`getProperty`

Sound (object)

The Sound object allows you to set and control sounds in a particular movie clip instance, or for the global Timeline, if you do not specify a `target` when creating a new sound object. You must use the constructor `new Sound` to create an instance of the Sound object before calling the methods of the Sound object.

The Sound object is only supported for the Flash 5 Player.

Method summary for the Sound object

Method	Description
`attachSound`	Attaches the sound specified in the argument.
`getPan`	Returns the value of the previous setPan call.
`getTransform`	Returns the value of the previous setTransform call.
`getVolume`	Returns the value of the previous setVolume call.
`setPan`	Sets the left/right balance of the sound.
`setTransform`	Sets transform for a sound.
`setVolume`	Sets the volume level for a sound.
`start`	Starts playing a sound from the beginning or, optionally, from an offset point set in the argument.
`stop`	Stops the specified sound or all sounds currently playing.

Constructor for the Sound object

Syntax

```
new Sound();
new Sound(target);
```

Arguments

`target` The movie clip instance that the Sound object applies to. This argument is optional.

Description

Method; creates a new Sound object for a specified movie clip. If you do not specify a *`target`*, the Sound object controls all of the sounds in the global Timeline.

Player

Flash 5 or later.

Example

```
GlobalSound = new Sound();
MovieSound = new Sound(mymovie);
```

Sound.attachSound

Syntax

mySound`.attachSound("`*idName*`");`

Arguments

`idName` The name for the new instance of the sound. This is the same as the name entered for the identifier in the Symbol Linkage Properties dialog box. This argument must be enclosed in `" "` (quotation marks).

Description

Method; attaches the sound specified in the *`idName`* argument to the specified Sound object. The sound must be in the library of the current movie and specified for export in the Symbol Linkage Properties dialog box. You must call `Sound.start` to start playing the sound.

Player

Flash 5 or later.

See also

`Sound.start`

Sound.getPan

Syntax

mySound`.getPan();`

Arguments

None.

Description

Method; returns the pan level set in the last `setPan` call as an integer from -100 to 100. The pan setting controls the left-right balance of the current and future sounds in a movie.

This method is cumulative with the `setVolume` or `setTransform` methods.

Player

Flash 5 or later.

See also

`Sound.setPan`

`Sound.setTransform`

Sound.getTransform

Syntax

mySound`.getTransform();`

Arguments

None.

Description

Method; returns the sound transform information for the specified Sound object set with the last `setTransform` call.

Player

Flash 5 or later.

See also

`Sound.setTransform`

Sound.getVolume

Syntax

mySound`.getVolume();`

Arguments

None.

Description

Method; returns the sound volume level as an integer from 0 to 100, where 0 is off and 100 is full volume. The default setting is 100.

Player

Flash 5 or later.

See also

`Sound.setVolume`

Sound.setPan

Syntax

mySound.setPan(*pan*);

Arguments

pan An integer specifying the left-right balance for a sound. The range of valid values is -100 to 100, where -100 uses only the the left channel, 100 uses only the right channel, and 0 balances the sound evenly between the two channels.

Description

Method; determines how the sound is played in the left and right channels (speakers). For mono sounds, *pan* affects which speaker (left or right) the sound plays through.

This method is cumulative with the `setVolume` and `setTransform` methods, and calling this method deletes and updates previous `setPan` and `setTransform` settings.

Player

Flash 5 or later.

Example

The following example uses `setVolume` and `setPan` to control a sound object with the specified target "u2":

```
onClipEvent(mouseDown) {
// create a sound object and
s = new Sound(this);
// attach a sound in the library
s.attachSound("u2");
//set volume at 50%
s.setVolume(50);
//turn off the sound in the right channel
s.setPan(-100);
//start 30 seconds into the sound and play it 5 times
s.start(30, 5);
```

See also

`Sound.setTransform`
`Sound.setVolume`

Sound.setTransform

Syntax

`mySound.setTransform(soundTransformObject);`

Arguments

`soundTransformObject` An object created with the constructor for the generic Object object.

Description

Method; sets the sound transform information for a Sound object. This method is cumulative with the `setVolume` and `setPan` methods, and calling this method deletes and updates any previous `setPan` or `setVolume` settings. This call is for expert users who want to add interesting effects to sounds.

Sounds use a considerable amount of disk space and memory. Because stereo sounds use twice as much data as mono sounds, it's generally best to use 22-Khz 6-bit mono sounds. You can use the `setTransform` method to play mono sounds as stereo, play stereo sounds as mono, and to add interesting effects to sounds.

The `soundTransformObject` argument is an object that you create using the constructor method of the generic Object object with parameters specifying how the sound is distributed to the left and right channels (speakers).

The parameters for the `soundTransformObject` are as follows:

`ll` A percentage value specifying how much of the left input to play in the left speaker (-100 to 100).

`lr` A percentage value specifying how much of the right input to play in the left speaker (-100 to 100).

`rr` A percentage value specifying how much of the right input to play in the right speaker (-100 to 100).

`rl` A percentage value specifying how much of the left input to play in the right speaker (-100 to 100).

The net result of the parameters is represented by the following formula:

leftOutput = left input * ll + right input * lr

rightOutput = right lnput * rr + left input * rl

The values for left input or right input are determined by the type (stereo or mono) of sound in your movie.

Stereo sounds divide the sound input evenly between the left and right speakers and have the following transform settings by default:

```
ll = 100
lr = 0
rr = 100
rl = 0
```

Mono sounds play all sound input in the left speaker and have the following transform settings by default:

```
ll = 100
lr = 100
rr = 0
rl = 0
```

Player

Flash 5 or later.

Example

The following example creates a sound transform object that plays both the left and right channels in the left channel:

```
mySoundTransformObject = new Object
mySoundTransformObject.ll = 100
mySoundTransformObject.lr = 100
mySoundTransformObject.rr = 0
mySoundTransformObject.rl = 0
```

In order to apply the sound transform object to a Sound object, you need to pass the object to the Sound object using `setTransform` as follows:

```
mySound.setTransform(mySoundTransformObject);
```

The following are examples of settings that can be set using `setTransform`, but cannot be set using `setVolume` or `setPan`, even if combined.

This code plays both the left and right channels through the left channel:

```
mySound.setTransform(soundTransformObjectLeft);
```

In the above code, the *`soundTransformObjectLeft`* has the following parameters:

```
ll = 100
lr = 100
rr = 0
rl = 0
```

This code plays a stereo sound as mono:

```
setTransform(soundTransformObjectMono);
```

In the above code, the *`soundTransformObjectMono`* has the following parameters:

```
ll = 50
lr = 50
rr = 50
rl = 50
```

This code plays the left channel at half capacity and adds the rest of the left channel to the right channel:

```
setTransform(soundTransformObjectHalf);
```

In the above code, the `soundTransformObjectHalf` has the following parameters:

```
ll = 50
lr = 0
rr = 100
rl = 50
```

See also

Constructor for the Object object

Sound.setVolume

Syntax

`mySound``.setVolume(`*`volume`*`);`

Arguments

`volume` A number from 0 to 100 representing a volume level. 100 is full volume and 0 is no volume. The default setting is 100.

Description

Method; sets the volume for the Sound object.

This method is cumulative with the `setPan` and `setTransform` methods.

Player

Flash 5 or later.

Example

The following example sets volume to 50% and transfers the sound over time from the left speaker to the right speaker:

```
onClipEvent (load) {
   i = -100;
   s = new sound();
   s.setVolume(50);
}
onClipEvent (enterFrame) {
   S.setPan(i++);
}
```

See also

`Sound.setPan`
`Sound.setTransform`

Sound.start

Syntax

```
mySound.start();
mySound.start([secondOffset, loop]);
```

`secondOffset` An optional argument allowing you to start the sound playing at a specific point. For example, if you have a 30-second sound and want the sound to start playing in the middle, specify 15 for the `secondOffset` argument. The sound is not delayed 15 seconds, but rather starts playing at the 15-second mark.

`loop` An optional argument allowing you to specify the number of times the sound should loop.

Description

Method; starts playing the last attached sound from the beginning if no argument is specified, or starting at the point in the sound specified by the `secondOffset` argument.

Player

Flash 5 or later.

See also

`Sound.setPan`
`Sound.stop`

Sound.stop

Syntax

```
mySound.stop();
mySound.stop(["idName"]);
```

Arguments

`idName` An optional argument specifying a specific sound to stop playing. The `idName` argument must be enclosed in quotation marks(" ").

Description

Method; stops all sounds currently playing if no argument is specified, or just the sound specified in the `idName` argument.

Player

Flash 5 or later.

See also

`Sound.start`

_soundbuftime

Syntax

```
_soundbuftime = integer;
```

Arguments

integer The number of seconds before the movie starts to stream.

Description

Property (global); establishes the number of seconds of streaming sound to prebuffer. The default value is 5 seconds.

Player

Flash 4 or later.

startDrag

Syntax

```
startDrag(target);
startDrag(target,[lock]);
startDrag(target [,lock [,left , top , right, bottom]]);
```

Arguments

`target` The target path of the movie clip to drag.

`lock` A Boolean value specifying whether the draggable movie clip is locked to the center of the mouse position (`true`), or locked to the point where the user first clicked on the movie clip (`false`). This argument is optional.

`left`, `top`, `right`, `bottom` Values relative to the coordinates of the movie clip's parent that specify a constraint rectangle for the movie clip. These arguments are optional.

Description

Action; makes the `target` movie clip draggable while the movie is playing. Only one movie clip can be dragged at a time. Once a `startDrag` operation is executed, the movie clip remains draggable until explicitly stopped by a `stopDrag` action, or until a `startDrag` action for another movie clip is called.

Example

To create a movie clip that users can position in any location, attach the `startDrag` and `stopDrag` actions to a button inside the movie clip, as in the following:

```
on(press) {
   startDrag("",true);
}
on(release) {
   stopDrag();
}
```

See also

`stopDrag`

`_droptarget`

stop

Syntax

```
stop;
```

Arguments

None.

Description

Action; stops the movie that is currently playing. The most common use of this action is to control movie clips with buttons.

Player

Flash 3 or later.

stopAllSounds

Syntax

```
stopAllSounds();
```

Arguments

None.

Description

Action; stops all sounds currently playing in a movie without stopping the playhead. Sounds set to stream will resume playing as the playhead move over the frames they are in.

Player

Flash 3 or later.

Example

The following code could be applied to a button that, when clicked, stops all sounds in the movie:

```
on(release) {
   stopAllSounds();
}
```

See also

`Sound (object)`

stopDrag

Syntax

```
stopDrag();
```

Arguments

None.

Description

Action; stops the current drag operation.

Player

Flash 4 or later.

Example

This statement stops the drag action on the instance `mc` when the user releases the mouse button:

```
on(press) {
   startDrag("mc");
}
on(release) {
   stopdrag();
}
```

See also

`startDrag`
`_droptarget`

String (function)

Syntax

`String(`*`expression`*`);`

Arguments

`expression` The number, Boolean, variable, or object to convert to a string.

Description

Function; returns a string representation of the specified argument as follows:

If *`x`* is Boolean, the return string is `true` or `false`.

If *`x`* is a number, the return string is a decimal representation of the number.

If *`x`* is a string, the return string is `x`.

If *`x`* is an object, the return value is a string representation of the object generated by calling the string property for the object, or by calling `object.toString` if no such property exists.

If *`x`* is a movie clip, the return value is the target path of the movie clip in slash (/) notation.

If *`x`* is undefined, the return value is an empty string.

Player

Flash 3 or later.

See also

```
Object.toString
Number.toString
String (object)
" " (string delimiter)
```

" " (string delimiter)

Syntax

`"text"`

Arguments

text Any text.

Description

String delimiter; when used before and after a string, quotes indicate that the string is a literal—not a variable, numerical value, or other ActionScript element.

Player

Flash 4 or later.

Example

This statement uses quotes to indicate that the string "Prince Edward Island" is a literal string, and not the value of a variable:

```
province = "Prince Edward Island"
```

See also

`String (object)`
`String (function)`

String (object)

The String object is a wrapper for the string primitive data type, which allows you to use the methods and properties of the String object to manipulate primitive string value types. You can convert the value of any object into a string using the `String()` function.

All of the methods of the String object, except for `concat`, `fromCharCode`, `slice`, and `substr`, are generic. This means the methods themselves call `this.toString` before performing their operations, and you can use these methods with other non-String objects.

You can call any of the methods of the String object using the constructor method `new String` or using a string literal value. If you specify a string literal, the ActionScript interpreter automatically converts it to a temporary String object, calls the method, and then discards the temporary String object. You can also use the `String.length` property with a string literal.

It is important that you do not confuse a string literal with an instance of the String object. In the following example the first line of code creates the string literal `s1`, and the second line of code creates an instance of the String object `s2`.

```
s1 = "foo"
s2 = new String("foo")
```

It is recommended that you use string literals unless you specifically need to use a String object, as String objects can have counterintuitive behavior.

Method summary for String object

Method	Description
`charAt`	Returns a number corresponding to the placement of the character in the string.
`charCodeAt`	Returns the value of the character at the given index as a 16-bit integer between 0 and 65535.
`concat`	Combines the text of two strings and returns a new string.
`fromCharCode`	Returns a string made up of the characters specified in the arguments.
`indexOf`	Searches the string and returns the index of the value specified in the arguments. If value occurs more than once, the index of the first occurrence is returned. If value is not found, -1 is returned.
`lastIndexOf`	Returns the last occurrence of substring within the string that appears before the start position specified in the argument, or -1 if not found.
`slice`	Extracts a section of a string and returns a new string.
`split`	Splits a String object into an array of strings by separating the string into substrings.
`substr`	Returns a specified number of characters in a string, beginning at the location specified in the argument.
`substring`	Returns the characters between two indexes, specified in the arguments, into the string.
`toLowerCase`	Converts the string to lowercase and returns the result.
`toUpperCase`	Converts the string to uppercase and returns the result.

Property summary for the String object

Property	Description
`length`	Returns the length of the string.

Constructor for the String object

Syntax

```
new String(value);
```

Arguments

value The initial value of the new String object.

Description

Constructor; creates a new String object.

Player

Flash 5 or later.

See also

```
String (function)
" " (string delimiter)
```

String.charAt

Syntax

```
myString.charAt(index);
```

Arguments

index The number of the character in the string to be returned.

Description

Method; returns the character specified by the argument *index*. The index of the first character in a string is 0. If *index* is not a number from 0 to `string.length` - 1, an empty string is returned.

Player

Flash 5 or later.

String.charCodeAt

Syntax

`myString.charCodeAt(index);`

Arguments

`index` The number of the character for which the value is retrieved.

Description

Method; returns the value of the character specified by `index`. The returned value is a 16-bit integer from 0 to 65535.

This method is similar to `string.charAt` except that the returned value is for the character at a specific location, instead of a substring containing the character.

Player

Flash 5 or later.

String.concat

Syntax

`myString.concat(value1,...valueN);`

Arguments

`value1,...valueN` Zero or more values to be concatenated.

Description

Method; combines the specified values and returns a new string. If necessary, each `value` argument is converted to a string and appended, in order, to the end of the string.

Player

Flash 5 or later.

String.fromCharCode

Syntax

`myString.fromCharCode(c1,c2,...cN);`

Arguments

`c1,c2,...cN` The characters to be made into a string.

Description

Method; returns a string made up of the characters specified in the arguments.

Player

Flash 5 or later.

String.indexOf

Syntax

```
myString.indexOf(value);
myString.index of (value, start);
```

Arguments

value An integer or string specifying the substring to be searched for within *myString*.

start An integer specifying the starting point of the substring. This argument is optional.

Description

Method; searches the string and returns the position of the first occurrence of the specified *value*. If the value is not found, the method returns -1.

Player

Flash 5 or later.

String.lastIndexOf

Syntax

```
myString.lastIndexOf(substring);
myString.lastIndexOf(substring, start);
```

Arguments

substring An integer or string specifying the string to be searched for.

start An integer specifying the starting point inside the substring. This argument is optional.

Description

Method; searches the string and returns the index of the last occurrence of `substring` found within the calling string. If *substring* is not found, the method returns -1.

Player

Flash 5 or later.

String.length

Syntax

`string.length`

Arguments

None.

Description

Property; returns the number of characters in the specified String object. The index of the last character for any string `x` is x.length-1.

Player

Flash 5 or later.

String.slice

Syntax

`myString.slice(start, end);`

Arguments

`start` A number specifying the index of the starting point for the slice. If *`start`* is a negative number, the starting point is determined from the end of the string, where -1 is the last character.

`end` A number specifying the index of the ending point for the slice. If *`end`* is not specified, the slice includes all characters from the start to the end of the string. If *`end`* is a negative number, the ending point is determined from the end of the string, where -1 is the last character.

Description

Method; extracts a slice, or substring, of the specified String object; then returns it as a new string without modifying the original String object. The returned string includes the *`start`* character and all characters up to (but not including) the *`end`* character.

Player

Flash 5 or later.

String.split

Syntax

`myString.split(delimiter);`

Arguments

`delimiter` The character used to delimit the string.

Description

Method; splits a String object by breaking the string wherever the specified `delimiter` argument occurs, and returns the substrings in an array. If no delimiter is specified, the returned array contains only one element—the string itself. If the delimiter is an empty string, each character in the String object becomes an element in the array.

Player

Flash 5 or later.

String.substr

Syntax

`myString.substr(start, length);`

Arguments

`start` An integer that indicates the position of the first character in the substring being created. If `start` is a negative number, the starting position is determined from the end of the string, where the -1 is the last character.

`length` The number of characters in the substring being created. If `length` is not specified, the substring includes all of the characters from the start to the end of the string.

Description

Method; returns the characters in a string from the index specified in the `start` argument through the number of characters specified in the `length` argument.

Player

Flash 5 or later.

String.substring

Syntax

`myString.substring(from, to);`

Arguments

from An integer that indicates the position of the first character in the substring being created. Valid values for *from* are 0 through `string.length` - 1.

to An integer that is 1+ the index of the last character in the substring being created. Valid values for *to* are 1 through `string.length`. If the *to* argument is not specified, the end of the substring is the end of the string. If *from* equals *to*, the method returns an empty string. If *from* is greater than *to*, the arguments are automatically swapped before the function executes.

Description

Method; returns a string consisting of the characters between the points specified by the *from* and *to* arguments.

Player

Flash 5 or later.

String.toLowerCase

Syntax

`myString.toLowerCase();`

Arguments

None.

Description

Method; returns a copy of the String object, with all of the uppercase characters converted to lowercase.

Player

Flash 5 or later.

String.toUpperCase

Syntax

`myString.toUpperCase();`

Arguments

None.

Description

Method; returns a copy of the String object, with all of the lowercase characters converted to uppercase.

Player

Flash 5 or later.

substring

Syntax

`substring(string, index, count);`

Arguments

`string` The string from which to extract the new string.

`index` The number of the first character to extract.

`count` The number of characters to include in the extracted string, not including the index character.

Description

String function; extracts part of a string.

Player

Flash 4 or later. This function has been deprecated in Flash 5.

See also

`String.substring`

_target

Syntax

instancename._target

Arguments

instancename The name of a movie clip instance.

Description

Property (read-only); returns the target path of the movie clip instance specified in the *instancename* argument.

Player

Flash 4 or later.

targetPath

Syntax

targetpath(*movieClipObject*);

Arguments

movieClipObject Reference (for example, _root or _parent) to the movie clip for which the target path is being retrieved.

Description

Function; returns a string containing the target path of *movieClipObject*. The target path is returned in dot notation. To retrieve the target path in slash notation, use the _target property.

Player

Flash 5 or later.

Example

The following examples are equivalent. The first example uses dot notation, and the second example uses slash notation.

```
targetPath (Board.Block[index*2+1]) {
play();
}
```

Is equivalent to:

```
tellTarget ("Board/Block:" + (index*2+1)) {
play();
}
```

See also

eval

tellTarget

Syntax

```
tellTarget(target) {
statement;
}
```

Arguments

`target` A target path string specifying the Timeline to be controlled.

`statement` Instructions applied to the targeted Timeline.

Description

Action; applies the instructions specified in the *`statements`* argument to the Timeline specified in the *`target`* argument. The `tellTarget` action is useful for navigation controls. Assign `tellTarget` to buttons that stop or start movie clips elsewhere on the Stage. You can also make movie clips go to a particular frame in that clip. For example, you might assign `tellTarget` to buttons that stop or start movie clips on the Stage or prompt movie clips to jump to a particular frame.

The `tellTarget` action is very similar to the `with` action, except that `with` takes a movie clip or other object as a *`target`*, and `tellTarget` requires a target path to a movie clip and cannot control objects.

Player

Flash 3 or later. This action is deprecated in Flash 5; use of the `with` action is recommended.

Example

This `tellTarget` statement controls the movie clip instance `ball` on the main Timeline. Frame 1 of the movie clip is blank and has a `stop` action so that it isn't visible on the Stage. When the button with the following action is clicked, `tellTarget` tells the playhead in the movie clip `ball` to go to frame 2 and play the animation that starts there.

```
on(release) {
   tellTarget("ball") {
      gotoAndPlay(2);
   }
}
```

See also

`with`

this

Syntax

```
this
```

Arguments

None.

Description

Keyword; references an object or movie clip instance. The keyword `this` has the same purpose and function in ActionScript as it does in JavaScript, with some additional functionality. In ActionScript, when a script executes, `this` references the movie clip instance that contains the script. When used with a method invocation, `this` contains a reference to the object that contains the executed method.

Player

Flash 5 or later.

Example

In the following example, the keyword `this` references the Circle object:

```
function Circle(radius){
   this.radius = radius;
   this.area = math.PI * radius * radius;
}
```

In the following statement assigned to a frame, the keyword `this` references the current movie clip:

```
//sets the alpha property of the current movie clip to 20.
this._alpha = 20;
```

In the following statement inside an `onClipEvent` handler, the keyword `this` references the current movie clip:

```
//when the movie clip loads, a startDrag operation is initiated
for the current movie clip.

onClipEvent (load) {
   startDrag (this, true);
}
```

See also

`new`

toggleHighQuality

Syntax

```
toggleHighQuality();
```

Arguments

None.

Description

Action; turns antialiasing on and off in the Flash Player. Antialiasing smooths the edges of objects and slows down the movie playback. The `toggleHighQuality` action affects all movies in the Flash Player.

Player

Flash 2 or later.

Example

The following code could be applied to a button that, when clicked, would toggle antialiasing on and off:

```
on(release) {
   toggleHighQuality();
}
```

See also

`_quality`
`_highquality`

_totalframes

Syntax

`instancename``._totalframes`

Arguments

`instancename` The name of the movie clip to evaluate.

Description

Property (read-only); evaluates the movie clip specified in the *`instancename`* argument and returns the total number of frames in the movie.

Player

Flash 4 or later.

trace

Syntax

`trace(`*`expression`*`);`

Arguments

`expression` A statement to evaluate. When you test the movie, the results of the *`expression`* argument are displayed in the Output window.

Description

Action; evaluates the *`expression`* and displays the results in the Output window in test-movie mode.

Use `trace` to record programming notes or to display messages in the Output window while testing a movie. Use the *`expression`* parameter to check if a condition exists, or to display values in the Output window. The `trace` action is similar to the `alert` function in JavaScript.

Player

Flash 4 or later.

Example

This example is from a game in which a draggable movie clip instance named `rabbi` must be released on a specific target. A conditional statement evaluates the `_droptarget` property and executes different actions depending on where `rabbi` is released. The `trace` action is used at the end of the script to evaluate the location of the `rabbi` movie clip, and display the results in the Output window. If `rabbi` doesn't behave as expected (for example, if it snaps to the wrong target), the values sent to the Output window by the `trace` action will help you determine the problem in the script.

```
on(press) {
rabbi.startDrag();
}
on(release) {
if(eval(_droptarget) != target) {
     rabbi._x = rabbi_x;
     rabbi._y = rabbi_y;
} else {
rabbi_x = rabbi._x;
rabbi_y = rabbi._y;
target = "_root.pasture";
}
trace("rabbi_y = " + rabbi_y);
trace("rabbi_x = " + rabbi_x);
stopDrag();
}
```

typeof

Syntax

```
typeof(expression);
```

Arguments

expression A string, movie clip, object, or function.

Description

Operator; a unary operator placed before a single argument. Causes Flash to evaluate *expression*; the result is a string specifying whether the expression is a string, movie clip, object, or function.

Player

Flash 5 or later.

unescape

Syntax

```
unescape(x);
```

Arguments

x A string with hexadecimal sequences to escape.

Description

Top-level function; evaluates the argument *x* as a string, decodes the string from a URL-encoded format (converting all hexadecimal sequences to ASCII characters), and returns the string.

Player

Flash 5 or later.

Example

The following example illustrates the escape-to-unescape conversion process.

```
escape("Hello{[World]}");
```

The escaped result is as follows:

```
("Hello%7B%5BWorld%5D%7D');
```

Use `unescape` to return to the original format:

```
unescape("Hello%7B%5BWorld%5D%7D")
```

The result is as follows:

```
Hello{[World]}
```

unloadMovie

Syntax

`unloadMovie(location);`

Arguments

`location` The depth level or target movie clip from which to unload the movie.

Description

Action; removes a movie from the Flash Player that was previously loaded using the `loadMovie` action.

Player

Flash 3 or later.

Example

The following example unloads the main movie, leaving the Stage blank:

```
unloadMovie(_root);
```

The following example unloads the movie at level 15, when the user clicks the mouse:

```
on(press) {
   unloadMovie(_level15);
}
```

See also

`loadMovie`

updateAfterEvent

Syntax

`updateAfterEvent(movie clip event);`

Arguments

`movie clip event` You can specify one of the following values as a movie clip event:

- `mouseMove` The action is initiated every time the mouse is moved. Use the `_xmouse` and `_ymouse` properties to determine the current mouse position.
- `mouseDown` The action is initiated if the left mouse button is pressed.
- `mouseUp` The action is initiated if the left mouse button is released.
- `keyDown` The action is initiated when a key is pressed. Use the `Key.getCode` method to retrieve information about the last key pressed.
- `keyUp` The action is initiated when a key is released. Use the `key.getCode` method to retrieve information about the last key pressed.

Description

Action; updates the display (independent of the frames per second set for the movie) after the clip event specified in the arguments has completed. This action is not listed in the Flash Actions panel. Using `updateAfterEvent` with drag actions that specify the `_x` and `_y` properties during the mouse move allows objects to drag smoothly without a flickering screen effect.

Player

Flash 5 or later.

See also

`onClipEvent`

_url

Syntax

`instancename``._url`

Arguments

`instancename` The target movie clip.

Description

Property (read only); retrieves the URL of the SWF file from which the movie clip was downloaded.

Player

Flash 4 or later?

var

Syntax

`var variableName1 [= value1] [...,variableNameN [=valueN]];`

Arguments

`variableName` The name of the variable to declare.

`value` The value being assigned to the variable.

Description

Action; used to declare local variables. If you declare local variables inside a function, the variables are defined for the function and expire at the end of the function call. If variables are not declared inside a block, but the action list was executed with a `call` action, the variables are local and expire at the end of the current list. If variables are not declared inside a block and the current action list was not executed with the call action, the variables are not local.

Player

Flash 5 or later.

_visible

Syntax

```
instancename._visible
instancename._visible = Boolean;
```

Arguments

`Boolean` Enter a *`true`* or *`false`* value to specify whether the movie clip is visible.

Description

Property; determines whether or not the movie specified by the *`instancename`* argument is visible. Movie clips that are not visible (property set to `false`) are disabled. For example, a button in a movie clip with the `_visible` property set to `false` cannot be clicked.

Player

Flash 4 or later.

void

Syntax

```
void (expression);
```

Arguments

expression An expression of any value.

Description

Operator; a unary operator that discards the *expression* value and returns an undefined value. The `void` operator is often used to evaluate a URL in order to test for side effects without displaying the evaluated expression in the browser window. The `void` operator is also used in comparisons using the == operator to test for undefined values.

Player

Flash 5 or later.

while

Syntax

```
while(condition) {
statement(s);
}
```

Arguments

condition The statement that is reevaluated each time the `while` action is executed. If the statement evaluates to `true`, the expression in the *statement(s)* is run.

statement(s) The expression to run if the condition evaluates to `true`.

Description

Action; runs a statement or series of statements repeatedly in a loop as long as the condition argument is `true`. At the end of each `while` action, Flash restarts the loop by retesting the condition. If the condition is `false` or equal to 0, Flash skips to the first statement after the `while` action.

Looping is commonly used to perform an action while a counter variable is less than a specified value. At the end of each loop, the counter is incremented until the threshold value is reached, the *condition* is no longer `true`, and the loop ends.

Player

Flash 4 or later.

Example

This example duplicates five movie clips on the Stage, each with a randomly generated *x* and *y* position, `xscale` and `yscale`, and `_alpha` property to achieve a scattered effect. The variable `foo` is initialized with the value 0. The *`condition`* argument is set so that the `while` loop will run five times, or as long as the value of the variable `foo` is less than 5. Inside the `while` loop, a movie clip is duplicated and `setProperty` is used to adjust the various properties of the duplicated movie clip. The last statement of the loop increments `foo` so that when the value reaches 5, the *`condition`* argument evaluates to `false`, and the loop will not be executed.

```
on(release) {
   foo = 0;
   while(foo < 5) {
      duplicateMovieClip("/flower", "mc" + foo, foo);
      setProperty("mc" + foo, _x, random(275));
      setProperty("mc" + foo, _y, random(275));
      setProperty("mc" + foo, _alpha, random(275));
      setProperty("mc" + foo, _xscale, random(200));
      setProperty("mc" + foo, _yscale, random(200));
      foo = foo + 1;
   }
}
```

See also

`do...while`
`continue`

_width

Syntax

instancename._width
instancename._width =*value*;

Arguments

value The width of the movie in pixels.

instancename An instance name of a movie clip for which the _width property is to be set or retrieved.

Description

Property; sets the width of the movie. In previous versions of Flash, _height and _width were read-only properties; in Flash 5 they can be set as well as retrieved.

Player

Flash 4 as a read-only property. In Flash 5 or later, this property can be set as well as retrieved.

Example

The following code example sets the height and width properties of a movie clip when the user clicks the mouse:

```
onclipEvent(mouseDown) {
   _width=200;
   _height=200;
}
```

See also

_height

with

Syntax

```
with (object) {
statement(s);
}
```

Arguments

`object` An instance of an ActionScript object or movie clip.

`statement(s)` An action or group of actions enclosed in curly braces.

Description

Action; temporarily changes the scope (or target path) used for evaluating expressions and actions in the `statement(s)`. After the `with` action executes, the scope chain is restored to its original state.

The `object` becomes the context in which the properties, variables, and functions are read. For example, if `object` is `myArray`, and two of the properties specified are `length` and `concat`, those properties are automatically read as `myArray.length` and `myArray.concat`. In another example, if `object` is `state.california`, it is as if any actions or statements inside the `with` action were called from inside the `california` instance.

To find the value of an identifier in the `statement(s)`, ActionScript starts at the beginning of the scope chain specified by the `object` and searches for the identifier at each level of the scope chain, in a specific order.

The scope chain used by the `with` action to resolve identifiers starts with the first item in the following list and continues to the last, as follows:

- `object` referenced by innermost `with` action
- `object` referenced by outermost `with` action
- Activation object (A temporary object that is automatically created when a function is called that holds the local variables called in the function.)
- Movie clip containing currently executing script
- Global object (predefined objects such as Math, String)

In Flash 5 the `with` action replaces the deprecated `tellTarget` action. You are encouraged to use `with` instead of `tellTarget` because it is a standard ActionScript extension to the ECMA-262 standard. The principal difference between the `with` and `tellTarget` actions is that `with` takes a reference to a movie clip or other object as its argument, while `tellTarget` takes a target path string identifying a movie clip, and cannot be used to target objects.

To set a variable inside a `with` action, the variable must have been delared outside the `with` action or you must enter the full path to the Timeline on which you want the variable to live. If you set a variable in a `with` action without having declared it, the `with` action will look for the value according to the scope chain. If the variable doesn't already exist, the new value will be set on the Timeline from which the `with` action was called.

Example

The following example sets the *x* and *y* properties of the `someOtherMovieClip` instance, and then instructs `someOtherMovieClip` to go to frame 3 and stop:

```
with (someOtherMovieClip) {
   _x = 50;
   _y = 100;
   gotoAndStop(3);
}
```

The following code snippet is how you would write the preceding code without using a `with` action:

```
someOtherMovieClip._x = 50;
someOtherMovieClip._y = 100;
someOtherMovieClip.gotoAndStop(3);
```

This code could also be written using the `tellTarget` action:

```
tellTarget ("someOtherMovieClip") {
   _x = 50;
   _y = 100;
   gotoAndStop(3);
}
```

The `with` action is useful for accessing multiple items in a scope chain list simultaneously. In the following example, the built-in `Math` object is placed at the front of the scope chain. Setting `Math` as a default object resolves the identifiers `cos`, `sin`, and `PI` to `Math.cos`, `Math.sin`, and `Math.PI`, respectively. The identifiers `a`, `x`, `y`, and `r` are not methods or properties of the `Math` object, but since they exist in the object activation scope of the function `polar`, they resolve to the corresponding local variables.

```
function polar(r){
   var a, x, y
   with (Math) {
      a = PI * r * r
      x = r * cos(PI)
      y = r * sin(PI/2)
}
trace("area = " +a)
trace("x = " + x)
trace("y = " + y)
}
```

You can use nested `with` actions to access information in multiple scopes. In the following example, the instance `fresno` and the instance `salinas` are children of the instance `california`. The statement sets the `_alpha` values of `fresno` and `salinas` without changing the `_alpha` value of `california`.

```
with (california){
   with (fresno){
      _alpha = 20;
   }
   with (salinas){
      _alpha = 40;
   }
}
```

See also

`tellTarget`

_x

Syntax

```
instancename._x
instancename._x = integer
```

Arguments

`integer` The local *x* coordinate of the movie.

`instancename` The name of a movie clip instance.

Description

Property; sets the *x* coordinate of movie relative to the local coordinates of the parent movie clip. If a movie clip is in the main Timeline, then its coordinate system refers to the upper left corner of the Stage as (0, 0). If the move clip is inside another movie clip that has transformations, the movie clip is in the local coordinate system of the enclosing movie clip. Thus, for a movie clip rotated 90° counterclockwise, the movie clip's children inherit a coordinate system that is rotated 90° counterclockwise. The movie clip's coordinates refer to the registration point position.

Player

Flash 3 or later.

See also

`_y`
`_xscale`

XML (object)

Use the methods and properties of the XML object to load, parse, send, build, and manipulate XML document trees.

You must use the constructor `new XML()` to create an instance of the XML object before calling any of the methods of the XML object.

XML is supported by Flash 5 or later versions of the Flash Player.

Method summary for the XML object

Method	Description
`appendChild`	Appends a node to the end of the specified object's child list.
`cloneNode`	Clones the specified node and, optionally, recursively clones all children.
`createElement`	Creates a new XML element.
`createTextNode`	Creates a new XML text node.
`hasChildNodes`	Returns `true` if the specified node has child nodes; otherwise, returns `false`.
`insertBefore`	Inserts a node in front of an existing node in the specified node's child list.
`load`	Loads a document (specified by the XML object) from a URL.
`onLoad`	A callback function for `load` and `sendAndLoad`.
`parseXML`	Parses an XML document into the specified XML object tree.
`removeNode`	Removes the specified node from its parent.
`send`	Sends the specified XML object to a URL.
`sendAndLoad`	Sends the specified XML object to a URL and loads the server response into another XML object.
`toString`	Converts the specified node and any children to XML text.

Property summary for the XML object

Property	Description
`docTypeDecl`	Sets and returns information about an XML document's DOCTYPE declaration.
`firstChild`	References the first child in the list for the specified node.
`lastChild`	References the last child in the list for the specified node.
`loaded`	Checks if the specified XML object has loaded.
`nextSibling`	References the next sibling in the parent node's child list.
`nodeName`	Returns the tag name of an XML element.
`nodeType`	Returns the type of the specified node (XML element or text node).
`nodeValue`	Returns the text of the specified node if the node is a text node.
`parentNode`	References the parent node of the specified node.
`previousSibling`	References the previous sibling in the parent node's child list.
`status`	Returns a numeric status code indicating the success or failure of an XML document parsing operation.
`xmlDecl`	Sets and returns information about an XML document's document declaration.

Collections summary for the XML object

Method	Description
`attributes`	Returns an associative array containing all of the attributes of the specified node.
`childNodes`	Returns an array containing references to the child nodes of the specified node.

Constructor for the XML object

Syntax

```
new XML();
new XML(source);
```

Arguments

`source` The XML document parsed to create the new XML object.

Description

Constructor; creates a new XML object. You must use the constructor method to create an instance of the XML object before calling any of the XML object methods.

The first syntax constructs a new, empty XML object.

The second syntax constructs a new XML object by parsing the XML document specified in the `source` argument, and populates the newly created XML object with the resulting XML document tree.

Note: The `createElement` and `createTextnode` methods are the 'constructor' methods for creating the elements and text nodes in an XML document tree.

Player

Flash 5 or later.

Example

The following example creates an new empty XML object:

```
myXML = new XML();
```

See also

`XML.createTextNode`
`XML.createElement`

XML.appendChild

Syntax

myXML.appendChild(*childNode*);

Arguments

childNode The child node to be added to the specified XML object's child list.

Description

Method; appends the specified child node to the XML object's child list. The appended child node is placed in the tree structure once removed from its existing parent node, if any.

Player

Flash 5 or later.

Example

The following example clones the last node from `doc1` and appends it to `doc2`:

```
doc1 = new XML(src1);
doc2 = new XML();
node = doc1.lastChild.cloneNode(true);
doc2.appendChild(node);
```

XML.attributes

Syntax

myXML`.attributes;`

Arguments

None.

Description

Collection (read-write); returns an associative array containing all attributes of the specified XML object.

Player

Flash 5 or later.

Example

The following example writes the names of the XML attributes to the Output window:

```
str = "<mytag name=\"Val\"> intem </mytage>";
doc = new XML(str);
y = doc.fristChild.attributes.name;
   trace (y);
doc.firstChild.attributes.order = "first";
z = doc.firstChild.attributes.order
   trace(z);
```

The following is written to the Output window:

```
Val
First
```

XML.childNodes

Syntax

myXML`.childNodes;`

Arguments

None.

Description

Collection (read-only); returns an array of the specified XML object's children. Each element in the array is a reference to an XML object that represents a child node. This is a read-only property and cannot be used to manipulate child nodes. Use the methods `appendChild`, `insertBefore`, and `removeNode` to manipulate child nodes.

This collection is undefined for text nodes (nodeType == 3).

Player

Flash 5 or later.

XML.cloneNode

Syntax

myXML.`cloneNode(`*deep*`);`

Arguments

deep Boolean value specifying whether the children of the specified XML object are recursively cloned.

Description

Method; constructs and returns a new XML node of the same type, name, value, and attributes as the specified XML object. If *deep* is set to `true`, all child nodes are recursively cloned, resulting in an exact copy of the original object's document tree.

Player

Flash 5 or later.

XML.createElement

Syntax

myXML.`createElement(`*name*`);`

Arguments

name The tag name of the XML element being created.

Description

Method; creates a new XML element with the name specified in the argument. The new element initially has no parent and no children. The method returns a reference to the newly created XML object representing the element. This method and `createTextNode` are the constructor methods for creating nodes for an XML object.

Player

Flash 5 or later.

XML.createTextNode

Syntax

`myXML.createTextNode(text);`

Arguments

text The text used to create the new text node.

Description

Method; creates a new XML text node with the specified text. The new node initially has no parent, and text nodes cannot have children. This method returns a reference to the XML object representing the new text node. This method and `createElement` are the constructor methods for creating nodes for an XML object.

Player

Flash 5 or later.

XML.docTypeDecl

Syntax

myXML.XMLdocTypeDecl;

Arguments

None.

Description

Property; sets and returns information about the XML document DOCTYPE declaration. After the XML text has been parsed into an XML object, the XML.docTypeDecl property of the XML object is set to the text of the XML document's DOCTYPE declaration. For example, <!DOCTYPE greeting SYSTEM "hello.dtd">. This property is set using a string representation of the DOCTYPE declaration, not an XML node object.

ActionScript's XML parser is not a validating parser. The DOCTYPE declaration is read by the parser and stored in the docTypeDecl property, but no DTD validation is performed.

If no DOCTYPE declaration was encountered during a parse operation, XML.docTypeDecl is set to undefined. XML.toString outputs the contents of XML.docTypeDecl immediately after the XML declaration stored in XML.xmlDecl, and before any other text in the XML object. If XML.docTypeDecl is undefined, no DOCTYPE declaration is output.

Player

Flash 5 or later.

Example

The following example uses XML.docTypeDecl to set the DOCTYPE declaration for an XML object.

```
myXML.docTypeDecl = "<!DOCTYPE greeting SYSTEM \"hello.dtd\">";
```

See also

XML.toString
XML.xmlDecl

XML.firstChild

Syntax

myXML.firstChild;

Arguments

None.

Description

Property (read-only); evaluates the specified XML object and references the first child in the parent node's children list. This property is `null` if the node does not have children. This property is undefined if the node is a text node. This is a read-only property and cannot be used to manipulate child nodes; use the methods `appendChild`, `insertBefore`, and `removeNode` to manipulate child nodes.

Player

Flash 5 or later.

See also

`XML.appendChild`
`XML.insertBefore`
`XML.removeNode`

XML.haschildNodes

Syntax

myXML.hasChildNodes();

Arguments

None.

Description

Method; evaluates the specified XML object and returns `true` if there are child nodes; otherwise, returns `false`.

Player

Flash 5 or later.

Example

The following example uses the information from the XML object in a user-defined function:

```
if (rootNode.hasChildNodes()) {
   myfunc (rootNode.firstChild);
}
```

XML.insertBefore

Syntax

`myXML.insertBefore(childNode, beforeNode);`

Arguments

`childNode` The node to be inserted.

`beforeNode` The node before the insertion point for the `childNode`.

Description

Method; inserts a new child node into the XML object's child list, before the `beforeNode`.

Player

Flash 5 or later.

XML.lastChild

Syntax

`myXML.lastChild;`

Arguments

None.

Description

Property (read-only); evaluates the XML object and references the last child in the parent node's child list. This method returns `null` if the node does not have children. This is a read-only property and cannot be used to manipulate child nodes; use the methods `appendChild`, `insertBefore`, and `removeNode` to manipulate child nodes.

Player

Flash 5 or later.

See also

`XML.appendChild`
`XML.insertBefore`
`XML.removeNode`

XML.load

Syntax

myXML`.load(`*url*`);`

Arguments

url The URL where the XML document to be loaded is located. The URL must be in the same subdomain as the URL where the movie currently resides.

Description

Method; loads an XML document from the specified URL, and replaces the contents of the specified XML object with the downloaded XML data. The load process is asynchronous; it does not finish immediately after the `load` method is executed. When `load` is executed, the XML object property `loaded` is set to `false`. When the XML data finishes downloading, the `loaded` property is set to `true`, and the `onLoad` method is invoked. The XML data is not parsed until it is completely downloaded. If the XML object previously contained any XML trees, they are discarded.

You can specify your own callback function in place of the `onLoad` method.

Player

Flash 5 or later.

Example

The following is a simple example using `XML.load`:

```
doc = new XML();
doc.load ("theFile.xml");
```

See also

`XML.onLoad`
`XML.loaded`

XML.loaded

Syntax

myXML`.loaded;`

Arguments

None.

Description

Property (read-only); determines whether the document loading process initiated by the `XML.load` call has completed. If the process completes successfully, the method returns `true`; otherwise, it returns `false`.

Player

Flash 5 or later.

Example

The following example uses `XML.loaded` in a simple script.

```
if (doc.loaded) {
   gotoAndPlay(4)
}
```

XML.nextSibling

Syntax

myXML`.nextSibling;`

Arguments

None.

Description

Property (read-only); evaluates the XML object and references the next sibling in the parent node's child list. This method returns `null` if the node does not have a next sibling node. This is a read-only property and cannot be used to manipulate child nodes. Use the methods `appendChild`, `insertBefore`, and `removeNode` to manipulate child nodes.

Player

Flash 5 or later.

See also

`XML.appendChild`
`XML.insertBefore`
`XML.removeNode`

XML.nodeName

Syntax

`myXML.nodeName;`

Arguments

None.

Description

Property; takes or returns the node name of the XML object. If the XML object is an XML element (`nodeType` == 1), `nodeName` is the name of the tag representing the node in the XML file. For example, `TITLE` is the nodeName of an HTML `TITLE` tag. If the XML object is a text node (`nodeType` == 3), the nodeName is `null`.

Player

Flash 5 or later.

See also

`XML.nodeType`

XML.nodeType

Syntax

`myXML.nodeType;`

Arguments

None.

Description

Property (read-only); takes or returns a `nodeType` value, where 1 is a XML element and 3 is a text node.

Player

Flash 5 or later.

See also

`XML.nodeValue`

XML.nodeValue

Syntax

myXML.nodeValue;

Arguments

None.

Description

Property; returns the node value of the XML object. If the XML object is a text node, the `nodeType` is 3, and the `nodeValue` is the text of the node. If the XML object is an XML element, it has a `null nodeValue` and is read-only.

Player

Flash 5 or later.

See also

`XML.nodeType`

XML.onLoad

Syntax

myXML.onLoad(*success*);

Arguments

success A boolean value indicating whether the XML object was successfully loaded with a `XML.load` or `XML.sendAndLoad` operation.

Description

Method; invoked by the Flash Player when an XML document is received from the server. If the XML document is received successfully, the *success* argument is `true`. If the document was not received, or if an error occurred in receiving the response from the server, the *success* argument is `false`. The default implementation of this method is not active. To override the default implementation, you must assign a function containing your own actions.

Player

Flash 5 or later.

Example

The following example creates a simple Flash movie for a simple e-commerce storefront application. We use the `sendAndLoad` method to transmit an XML element containing the user's name and password, and install an `onLoad` handler to handle the reply from the server.

```
var myLoginReply = new XML();
myLoginReply.onLoad = myOnLoad;
myXML.sendAndLoad("http://www.samplestore.com/login.cgi",
            myLoginReply);
function myOnLoad(success) {
   if (success) {
      if (e.firstChild.nodeName == "LOGINREPLY" &&
            e.firstChild.attributes.status == "OK") {
         gotoAndPlay("loggedIn")
      } else {
         gotoAndStop("loginFailed")
      }
   } else {
      gotoAndStop("connectionFailed")
   }
}
```

See also

`function`
`XML.load`
`XML.sendAndLoad`

XML.parentNode

Syntax

`myXML``.parentNode;`

Arguments

None.

Description

Property (read-only); references the parent node of the specified XML object, or returns `null` if the node has no parent. This is a read-only property and cannot be used to manipulate child nodes; use the methods `appendChild`, `insertBefore`, and `removeNode` to manipulate children.

Player

Flash 5 or later.

XML.parseXML

Syntax

myXML.parseXML(*source*);

Arguments

source The XML text to be parsed and passed to the specified XML object.

Description

Method; parses the XML text specified in the *source* argument, and populates the specified XML object with the resulting XML tree. Any existing trees in the XML object are discarded.

Player

Flash 5 or later.

XML.previousSibling

Syntax

myXML.previousSibling;

Description

Property (read-only); evaluates the XML object and references the previous sibling in the parent node's child list. Returns null if the node does not have a previous sibling node. This is a read-only property and cannot be used to manipulate child nodes; use the methods appendChild, insertBefore, and removeNode to manipulate child nodes.

Player

Flash 5 or later.

XML.removeNode

Syntax

myXML.removeNode();

Arguments

None.

Description

Method; removes the specified XML object from its parent.

Player

Flash 5 or later.

XML.send

Syntax

```
myXML.send(url);
myXML.send(url, window);
```

Arguments

`url` The destination URL for the specified XML object.

`window` The browser window to display data returned by the server: `_self` 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.

Description

Method; encodes the specified XML object into a XML document and sends it to the specified URL using the `POST` method.

Player

Flash 5 or later.

XML.sendAndLoad

Syntax

```
myXML.sendAndLoad(url,targetXMLobject);
```

Arguments

`url` The destination URL for the specified XML object. The URL must be in the same subdomain as the URL where the movie was downloaded from.

`targetXMLobject` An XML object created with the XML constructor method that will receive the return information from the server.

Description

Method; encodes the specified XML object into a XML document, sends it to the the specified URL using the `POST` method, downloads the server's response and then loads it into the *`targetXMLobject`* specified in the arguments. The server response is loaded in the same manner used by the `load` method.

Player

Flash 5 or later.

See also

`XML.load`

XML.status

Syntax

```
myXML.status;
```

Arguments

None.

Description

Property; automatically sets and returns a numeric value indicating whether an XML document was successfully parsed into an XML object. The following is a list of the numeric status codes and a description of each:

- 0 No error; parse completed successfully.
- -2 A CDATA section was not properly terminated.
- -3 The XML declaration was not properly terminated.
- -4 The DOCTYPE declaration was not properly terminated.
- -5 A comment was not properly terminated.
- -6 An XML element was malformed.
- -7 Out of memory.
- -8 An attribute value was not properly terminated.
- -9 A start-tag was not matched with an end-tag.
- -10 An end-tag was encountered without a matching start-tag.

Player

Flash 5 or later.

XML.toString

Syntax

```
myXML.toString();
```

Arguments

None.

Description

Method; evaluates the specified XML object, constructs a textural representation of the XML structure including the node, children, and attributes, and returns the result as a string.

For top-level XML objects (those created with the constructor), `XML.toString` outputs the document's XML declaration (stored in `XML.xmlDecl`), followed by the document's `DOCTYPE` declaration (stored in `XML.docTypeDecl`), followed by the text representation of all XML nodes in the object. The XML declaration is not output if `XML.xmlDecl` is undefined. The `DOCTYPE` declaration is not output if `XML.docTypeDecl` is undefined.

Player

Flash 5 or later.

Example

The following code is an example of the `XML.toString` method:

```
node = new XML("<h1>test</h1>");
trace(node.toString());
sends
<H1>test</H1>
to the output window
```

See also

`XML.xmlDecl`
`XML.docTypeDecl`

XML.xmlDecl

Syntax

```
myXML.xmlDecl;
```

Arguments

None.

Description

Property; sets and returns information about a document's XML declaration. After the XML document is parsed into an XML object, this property is set using the text of the document's XML declaration. This property is set using a string representation of the XML declaration, not an XML node object. If no XML declaration was encountered during a parse operation, the property is set to undefined. `XML.toString` outputs the contents of `XML.xmlDecl` before any other text in the XML object. If `XML.xmlDecl` contains the `undefined` type, no XML declaration is output.

Player

Flash 5 or later.

Example

The following example uses `XML.xmlDecl` to set the XML document declaration for an XML object:

```
myXML.xmlDecl = "<?xml version=\"1.0\" ?>";
```

See also

`XML.toString`
`XML.docTypeDecl`

XMLSocket (object)

The XMLSocket object implements client sockets that allow the computer running the Flash Player to communicate with a server computer identified by an IP address or domain name.

Using the XMLSocket object

To use the XMLSocket object, the server computer must run a daemon that understands the protocol used by the XMLSocket object. The protocol is as follows:

- XML messages are sent over a full-duplex TCP/IP stream socket connection.
- Each XML message is a complete XML document, terminated by a zero byte.
- An unlimited number of XML messages can be sent and received over a single XMLSocket connection.

The XMLSocket object is useful for client-server applications that require low latency, such as real-time chat systems. A traditional HTTP-based chat solution frequently polls the server and downloads new messages using an HTTP request. In contrast, an XMLSocket chat solution maintains an open connection to the server, which allows the server to immediately send incoming messages without a request from the client.

Setting up a server to communicate with the XMLSocket object can be challenging. If your application does not require real-time interactivity, use the `loadVariables` action, or Flash's HTTP-based XML server connectivity (`XML.load`, `XML.sendAndLoad`, `XML.send`), instead of the XMLSocket object.

To use the methods of the XMLSocket object, you must first use the constructor, `new XMLSocket`, to create a new XMLSocket object.

XMLSocket and security

Because the XMLSocket object establishes and maintains an open connection to the server, the following restrictions have been placed on the XMLSocket object for security reasons:

- The `XMLSocket.connect` method can connect only to TCP port numbers greater than or equal to 1024. One consequence of this restriction is that the server daemons that communicate with the XMLSocket object must also be assigned to port numbers greater than or equal to 1024. Port numbers below 1024 are often used by system services such as FTP, Telnet, and HTTP, thus barring the XMLSocket object from these ports. The port number restriction limits the possibility that these resources will be inappropriately accessed and abused.
- The `XMLSocket.connect` method can connect only to computers in the same subdomain where the SWF file (movie) resides. This restriction does not apply to movies running off a local disk. (This restriction is identical to the security rules for `loadVariables`, `XML.sendAndLoad`, and `XML.load`.)

Method summary for the XMLSocket object

Method	Description
`close`	Closes an open socket connection.
`connect`	Establishes a connection to the specified server.
`onClose`	A callback function that is invoked when an XMLSocket connection is closed.
`onConnect`	A callback function that is invoked when an XMLSocket connection is established.
`onXML`	A callback function that is invoked when an XML object arrives from the server.
`send`	Sends an XML object to the server.

Constructor for the XMLSocket object

Syntax

```
new XMLSocket();
```

Arguments

None.

Description

Constructor; creates a new XMLSocket object. The XMLSocket object is not initially connected to any server. You must call the `XMLSocket.connect` method to connect the object to a server.

Player

Flash 5 or later.

Example

```
myXMLSocket = new XMLSocket();
```

See also

`XMLSocket.connect`

XMLSocket.close

Syntax

`myXMLSocket.close();`

Arguments

None.

Description

Method; closes the connection specified by XMLSocket object.

Player

Flash 5 or later.

See also

`XMLSocket.connect`

XMLSocket.connect

Syntax

`myXMLSocket.connect(host, port);`

Arguments

`host` A fully qualified DNS domain name, or a IP address in the form *aaa.bbb.ccc.ddd.* You can also specify `null` to connect to the host server on which the movie resides.

`port` The TCP port number on the host used to establish a connection. The port number must be 1024 or higher.

Description

Method; establishes a connection to the specified Internet host using the specified TCP port (must be 1024 or higher), and returns `true` or `false` depending on whether a connection is successfully established. If you don't know the port number of your Internet host machine, contact your network administrator. If the Flash Netscape plug-in or ActiveX control is being used, the host specified in the argument must have the same subdomain as the host from where the movie was downloaded.

If you specify `null` for the *`host`* argument, the host contacted will be the host where the movie calling `XMLSocket.connect` resides. For example, if the movie was downloaded from http://www.yoursite.com, specifying `null` for the host argument is the same as entering the IP address for www.yoursite.com.

If `XMLSocket.connect` returns a value of `true`, the initial stage of the connection process is successful; later, the `XMLSocket.onConnect` method is invoked to determine whether the final connection succeeded or failed. If `XMLSocket.connect` returns `false`, a connection could not be established.

Player

Flash 5 or later.

Example

The following example uses `XMLSocket.connect` to connect to the host where the movie resides, and uses `trace` to display the return value indicating the success or failure of the connection:

```
function myOnConnect(success) {
   if (success) {
      trace ("Connection succeeded!")
  } else {
      trace ("Connection failed!")
   }
}
socket = new XMLSocket()
socket.onConnect = myOnConnect
if (!socket.connect(null, 2000)) {
   trace ("Connection failed!")
}
```

See also

`function`
`XMLSocket.onConnect`

XMLSocket.onClose

Syntax

myXMLSocket`.onClose();`

Arguments

None.

Description

Method; a callback function that is invoked only when an open connection is closed by the server. The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing your own actions.

Player

Flash 5 or later.

See also

`function`
`XMLSocket.onConnect`

XMLSocket.onConnect

Syntax

myXMLSocket`.onConnect(`*success*`);`

Arguments

success A Boolean value indicating whether a socket connection was successfully established (`true` or `false`).

Description

Method; a callback function invoked by the Flash Player when a connection request initiated through the `XMLSocket.connect` method has succeeded or failed. If the connection succeeded, the *success* argument is `true`; otherwise the *success* argument is `false`.

The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing your own actions.

Player

Flash 5 or later.

Example

The following example illustrates the process of specifying a replacement function for the `onConnect` method in a simple chat application.

The function controls which screen the users are taken to, depending on whether a connection is successfully established. If the connection is successfully established, users are taken to the main chat screen on the frame labeled `startChat`. If the connection is not successful, users go to a screen with troubleshooting information on the frame labeled `connectionFailed`.

```
function myOnConnect(success) {
   if (success) {
      gotoAndPlay("startChat")
   } else {
      gotoAndStop("connectionFailed")
   }
}
```

After creating the XMLSocket object using the constructor method, the script installs the `onConnect` method using the assignment operator:

```
socket = new XMLSocket()
socket.onConnect = myOnConnect
```

Finally, the connection is initiated. If `connect` returns `false`, the movie is sent directly to the frame labeled `connectionFailed`, and `onConnect` is never invoked. If `connect` returns `true`, the movie jumps to a frame labeled `waitForConnection`, which is the "Please wait" screen. The movie remains on the `waitForConnection` frame until the `onConnect` handler is invoked, which happens at some point in the future depending on network latency.

```
if (!socket.connect(null, 2000)) {
   gotoAndStop("connectionFailed")
} else {
   gotoAndStop("waitForConnection")
}
```

See also

`XMLSocket.connect`
`function`

XMLSocket.onXML

Syntax

`myXMLSocket.onXML(object);`

Argument

`object` An instance of the XML object containing a parsed XML document received from a server.

Description

Method; a callback function invoked by the Flash Player when the specified XML object containing an XML document arrives over an open XMLSocket connection. An XMLSocket connection may be used to transfer an unlimited number of XML documents between the client and the server. Each document is terminated with a zero byte. When the Flash Player receives the zero byte, it parses all of the XML received since the previous zero byte, or since the connection was established if this is the first message received. Each batch of parsed XML is treated as a single XML document and passed to the `onXML` method.

The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing actions that you define.

Player

Flash 5 or later.

Example

The following function overrides the default implementation of the `onXML` method in a simple chat application. The function `myOnXML` instructs the chat application to recognize a single XML element, `MESSAGE`, in the following format:

```
<MESSAGE USER="John" TEXT="Hello, my name is John!" />.
```

The `onXML` handler must first be installed in the XMLSocket object as follows:

```
socket.onXML = myOnXML;
```

The function `displayMessage` is assumed to be a user-defined function that displays the message received to the user.

```
function myOnXML(doc) {
   var e = doc.firstChild;
   if (e != null && e.nodeName == "MESSAGE") {
      displayMessage(e.attributes.user, e.attributes.text);
   }
}
```

See also

`function`

XMLSocket.send

Syntax

myXMLSocket.send(*object*);

Arguments

object An XML object or other data to transmit to the server.

Description

Method; converts the XML object or data specified in the *object* argument to a string and transmits it to the server, followed by a zero byte. If *object* is an XML object, the string is the XML textual representation of the XML object. The send operation is asynchronous; it returns immediately, but the data may be transmitted at a later time. The XMLSocket.send method does not return a value indicating whether the data was successfully transmitted.

If the *myXMLSocket* object is not connected to the server (using XMLSocket.connect), the XMLSocket.send operation will fail.

Player

Flash 5 or later.

Example

The following example illustrates how you could specify a user name and password to send the XML object myXML to the server:

```
var myXML = new XML();
var myLogin = myXML.createElement("login");
myLogin.attributes.username = usernameTextField;
myLogin.attributes.password = passwordTextField;
myXML.appendChild(myLogin);
myXMLSocket.send(myXML);
```

See also

XMLSocket.connect

_xmouse

Syntax

instancename._xmouse

Arguments

instancename The name of a movie clip instance.

Description

Property (read-only); returns the *x* coordinate of the mouse position.

Player

Flash 5 or later.

See also

Mouse (object)
_ymouse

_xscale

Syntax

```
instancename._xscale
instancename._xscale = percentage;
```

Arguments

percentage A percentage value specifying the percentage for horizontally scaling the movie. The default value is 100.

instancename The name of a movie clip instance.

Description

Property; determines the horizontal scale (*percentage*) of the movie clip as applied from the registration point of the movie clip. The default registration point is (0,0).

Scaling the local coordinate system affects the _x and _y property settings, which are defined in whole pixels. For example, if the parent movie clip is scaled to 50%, setting the _x property moves an object in the movie clip by half the number of pixels as it would if the movie were at 100%.

Player

Flash 4 or later.

See also

_xscale

_y

Syntax

```
instancename._y
instancename._y = integer;
```

Arguments

`integer` The local *y* coordinate of the movie clip.

`instancename` The name of a movie clip instance.

Description

Property; sets the *y* coordinate of movie relative to the local coordinates of the parent movie clip. If a movie clip is in the main Timeline, then its coordinate system refers to the upper left corner of the Stage as (0, 0). If the move clip is inside another movie clip that has transformations, the movie clip is in the local coordinate system of the enclosing movie clip. Thus, for a movie clip rotated 90° counterclockwise, the movie clip's children inherit a coordinate system that is rotated 90° counterclockwise. The movie clip's coordinates refer to the registration point position.

Player

Flash 3 or later.

See also

`_yscale`

_ymouse

Syntax

```
instancename._ymouse
```

Arguments

`instancename` The name of a movie clip instance.

Description

Property (read-only); indicates the *y* coordinate of the mouse position.

Player

Flash 5 or later.

See also

`Mouse (object)`
`_xmouse`

_yscale

Syntax

```
instancename._yscale
instancename._yscale = percentage;
```

Arguments

percentage A percentage value specifying the percentage for vertically scaling the movie. The default value is 100.

instancename The name of a movie clip instance.

Description

Property; sets the vertical scale (*percentage*) of the movie clip as applied from the registration point of the movie clip. The default registration point is (0,0).

Scaling the local coordinate system affects the `_x` and `_y` property settings, which are defined in whole pixels. For example, if the parent movie clip is scaled to 50%, setting the `_x` property moves an object in the movie clip by half the number of pixels as it would if the movie were at 100%.

Player

Flash 4 or later.

See also

`_x`
`_y`

APPENDIX A
Operator Precedence and Associativity

Operator List

This table lists all of Action Script operators and their associativity, from highest to lowest precedence.

Operator	Description	Associativity
	Highest Precedence	
+	Unary plus	Right to left
-	Unary minus	Right to left
~	Bitwise one's complement	Right to left
!	Logical NOT	Right to left
not	Logical NOT (Flash 4 style)	Right to left
++	Post-increment	Left to right
--	Post-decrement	Left to right
( )	Function call	Left to right
[ ]	Array element	Left to right
.	Structure member	Left to right
++	Pre-increment	Right to left
--	Pre-decrement	Right to left
new	Allocate object	Right to left
delete	Deallocate object	Right to left

Operator	Description	Associativity
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
+	Add	Left to right
add	String concatenation (formerly &)	Left to right
-	Subtract	Left to right
<<	Bitwise Left Shift	Left to right
>>	Bitwise Right Shift	Left to right
>>>	Bitwise Right Shift (Unsigned)	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
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
==	Equal	Left to right
!=	Not equal	Left to right
eq	Equal (string version)	Left to right
ne	Not equal (string version)	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
and	Logical AND (Flash 4)	Left to right

<table>
<tr><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>||</td><td>Logical OR</td><td>Left to right</td></tr>
<tr><td>or</td><td>Logical OR (Flash 4)</td><td>Left to right</td></tr>
<tr><td>?:</td><td>Conditional</td><td>Right to left</td></tr>
<tr><td>=</td><td>Assignment</td><td>Right to left</td></tr>
<tr><td>“*=, /=, %=, +=, -=, &=, |=, ^=, <<=, >>=, >>>=”</td><td>Compound assignment</td><td>Right to left</td></tr>
<tr><td>,</td><td>Multiple evaluation</td><td>Left to right</td></tr>
<tr><td colspan="3" align="center">Lowest Precedence</td></tr>
</table>

B

APPENDIX B
Keyboard Keys and Key Code Values

The following tables list all of the keys on a standard keyboard and the corresponding key code values that are used to identify the keys in ActionScript. For more information, see the description of the Key object in Chapter 7, "ActionScript Dictionary."

Letters A to Z and standard numbers 0 to 9

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

Letter or number key	Key code
M	77
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

Numeric keypad key	Key code
Numbpad 0	96
Numbpad1	97
Numbpad 2	98
Numbpad 3	99
Numbpad 4	100
Numbpad 5	101
Numbpad 6	102
Numbpad 7	103
Numbpad 8	104
Numbpad 9	105
Multiply	106
Add	107
Enter	108
Subtract	109
Decimal	110
Divide	111

Function keys

Function key	Key code
F1	112
F2	113
F3	114
F4	115
F5	116
F6	117
F7	118
F8	119
F9	120
F10	121
F11	122
F12	123

Other keys

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

<table>
<tr><th>Key</th><th>Key code</th></tr>
<tr><td>[{</td><td>219</td></tr>
<tr><td>\ |</td><td>220</td></tr>
<tr><td>] }</td><td>221</td></tr>
<tr><td>‘’ ‘</td><td>222</td></tr>
</table>

APPENDIX C
Error Messages

The following table contains a list of error messages returned by the Flash compiler. An explanation of each message is provided to aid you in troubleshooting your movie files.

Error message	Description
Property ‹property› does not exist	A property that does not exist was encountered. For example, `x = _green` is invalid, because there is no `_green` property.
Operator ‹operator› must be followed by an operand	An operator without an operand was encountered. For example, `x = 1 +` requires an operand after the + operator. An operator is followed by an invalid operand. For example, `trace(1+);` is syntactically incorrect.
Syntax error	This message is issued whenever a nonspecific syntax error is encountered.
Expected a field name after '.' operator	You must specify a valid field name when using the *`object.field`* syntax.

Expected ‹token›	An invalid or unexpected token was encountered. For example, in the syntax below, the token `foo` is not valid. The expected token is `while`.
	`do {`
	`trace (i)`
	`} foo (i < 100)`

Initializer list must be terminated by ‹terminator›	An object or array initializer list is missing the closing] or }.

Error message	Description
Identifier expected	An unexpected token was encountered in place of an identifier. In the example below, 3 is not a valid identifier. `var 3 = 4;`
The JavaScript '<construct>' construct is not supported	A JavaScript construct that is not supported by ActionScript was encountered. This message appears if any of the following JavaScript constructs are used: `void`, `switch`, `try`, `catch`, or `throw`.
Left side of assignment operator must be variable or property	An assignment operator was used, but the left side of the assignment was not a legal variable or property.
Statement block must be terminated by '}'	A group of statements was declared within curly braces, but the closing brace is missing.
Event expected	An `On(MouseEvent)` or `onClipEvent` handler was declared, but no event was specified, or an unexpected token was encountered where an event should appear.
Invalid event	The script contains an invalid mouse or clip event. For a list of valid mouse and clip events, see the On(MouseEvent) and OnClipEvent entries in the ActionScript dictionary chapter.
Key code expected	You need to specify a key code. See Appendix B for a list of key codes.
Invalid key code	The specified key code does not exist.
Trailing garbage found	The script or expression parsed correctly but contained additional trailing characters that could not be parsed
Illegal function	A named function declaration was used as an expression. Named function declarations must be statements. Valid: `function sqr (x) { return x * x; }` Invalid: `var v = function sqr (x) { return x * x; }`
Function name expected	The name specified for this function is not a valid function name.
Parameter name expected	A parameter (argument) name was expected in a function declaration, but an unexpected token was encountered.
'else' encountered without matching 'if'	An `else` statement was encountered, but no `if` appeared before it. You can use `else` only in conjunction with an `if` statement.

Error message	Description
Scene type error	The scene argument of a `gotoAndPlay`, `gotoAndStop`, or `ifFrameLoaded` action was of the wrong type. The scene argument must be a string constant.
Internal error	An internal error occurred in the ActionScript compiler. Please send the FLA file that generated this error to Macromedia, with detailed instructions on how to reproduce the message.
Hexadecimal digits expected after 0x	The sequence 0x was encountered, but the sequence was not followed by valid hexadecimal digits.
Error opening #include file	There was an error opening a file included with the `include` directive. The error may have occurred because the file was not present or because of a disk error.
Malformed #include directive	An `include` directive was not written correctly. An `include` directive must use the following syntax: `#include "`*`somefile`*`.as"`
Multi-line comment was not terminated	A multi-line comment started with `/*`is missing the closing `*/` tag.
String literal was not properly terminated	A string literal started with an opening quotation mark (single or double) is missing the closing quotation mark.
Function <function> takes <count> parameters	A function was called, but an unexpected number of parameters were encountered.
Property name expected in GetProperty	The `getProperty` function was called, but the second argument was not the name of a movie clip property.
Parameter <parameter> cannot be declared multiple times	A parameter name appeared multiple times in the parameter list of a function declaration. All parameter names must be unique.
Variable <variable> cannot be declared multiple times	A variable name appeared multiple times in a `var` statement. All variable names in a single `var` statement must be unique.
'on' handlers may not be nested within other 'on' handlers	An `on` handler was declared inside another `on` handler. All `on` handlers must appear at the top level of an action list.
Statement must appear within on handler	In the actions for a button instance, a statement was declared without a surrounding `on` block. All actions for a button instance must appear inside an `on` block.

Error message	Description
Statement must appear within onClipEvent handler	In the actions for a movie clip instance, a statement was declared without a surrounding `onClipEvent` block. All actions for a movie clip instance must appear inside an `onClipEvent` block.
Mouse events are permitted only for button instances	A button event handler was declared in a frame action list or a movie clip instance's action list. Button events are permitted only in the action lists of button instances.
Clip events are permitted only for movie clip instances	A clip event handler was declared in a frame action list or a button instance's action list. Clip events are permitted only in the action lists of movie clip instances.

INDEX

A

absolute target path 104
accessing
    methods 69
    object properties 57
actions 19
    assigning to control movies 114
    assigning to frames 35
    assigning to objects 33
    basic 79
    button parameters 36
    changing parameters 26
    compared to methods 112
    context-sensitive help 10
    deleting 26
    enabling simple 147
    exporting 30
    frame actions 35
    interactivity 79
    listed 58
    new features 8
    printing 30
    reordering 26
    repeating 61
    selecting 26
    targeting movie clips 111
    testing 33
    trace 156
    with target paths 59
Actions list, resizing 26
Actions panel 24
    categories 25
    displaying 24
    editing mode 24
    Normal Mode, Toolbox list 25
    options 29
ActionScript
    compared to JavaScript 7
    editing with text editor 27
    Flash 4 75
    Flash 4 compared to Flash 5 8
    JavaScript support 8
    new features 7
    scripting 12
    supported Flash 4 features 77
    syntax 37
    terminology 19
ActiveX controls 144
    displaying status 150
adding notes 41
animated symbols 45
arguments 19
    in parentheses 40
    passing to functions 66
array access operators 57
Ascii method 84
ASCII values 84
assigned functions 20
assignment operators 56
    compound 56
associativity, operators 52
asynchronous actions 129
attaching movie clips 118
attaching sounds 91
attachMovie method 111
attachMovieClip method 118
    arguments 118
attachSound method 90

B

balance (sound), controlling 93
behaviors 14
bitwise operators 55
Boolean values 44
    comparing 54

C

calling 45
    object methods 70
calling methods 45
capitalization 40
capturing keypresses 84
case-sensitivity
    keywords 40
    strings 43
CGI scripts, standard format 130
character sequences 43
characteristics 14
checklist, script 147
childNode 131
classes 14
    defined 19
clip parameters
    assigning 119
    defining 119
    setting 122
    setting Smart Clip 122
Clip Parameters panel, replacing with custom interface 123
collecting data 125
collisions
    between movie clip and Stage point 95
    between movie clips 95
    detecting 94
Color object 88
color values, setting 88
Colored Syntax command 31
combining operations 56
comments
    sample 41
    syntax 41
    syntax color 31, 41
communicating, between Timelines 102
communicating with the Flash Player 141
concatenating strings 43
conditional statements 17
conditions, checking for 60
constants
    defined 19
    syntax 42
construction functions, sample 14
constructor functions, sample 19
controlling movie clips, methods 111
controlling movies, requirements 108
controlling sound 90
Core JavaScript Guide 7
counters, repeating action with 61
creating, Smart Clips 119
creating objects 69
creating passwords 128
custom cursors, creating 80
custom functions 65
custom interface 119
    creating 123
    xch movie clip 124
custom objects 72

D

data types
    Boolean 44
    defined 19
    movie clips 45
    number 44
    objects 45
    rules 42
Debugger
    activating in Web browser 149
    display movie clips 150
    enabling 149
    Flash Debug Player 148
    movie properties 153
    password 149
    status bar 150
    using 148
    variables 150
    Watch list 152
declaring variables 49
deleting actions 26
detecting collisions 94
dialog boxes in forms 138
dimming Flash Player context menu 142
displaying, Flash Player context menu 142
dot operators 57

dot syntax 38
target paths 105
Drag Movie Clip action 117
dragging movie clips, evaluating 117
droptarget property 117
duplicateMovieClip action 102
duplicating movie clips 117
dynamic text 86

E

ECMA-262 specification 7
tellTarget action 112
editing modes
preference 28
switching 28
editing scripts
externally 29
mode 28
Enable Simple Buttons 147
Enable Simple Frame Actions 147
equality operators 56
errors
checking syntax 31
messages 32
name collision 48
escape sequences 43
European Computers Manufacturers Association (ECMA) 7
events, defined 19
executing application from projector 142
executing operators
by precedence 52
order by association 52
execution order 15
controlling 18
Expert Mode 27
calling function 63
exporting actions 30
exporting to Flash 4 77
expressions
about 51
assigning multiple variables 56
comparing values 53
defined 19
Extensible Markup Language 131
external editors 29

F

Flash 4 files, opening 75
Flash 5
creating Flash 4 content 77
Flash Debug Player 148
Flash Help, actions 10
Flash Player
communicating with 141
dimming context menu 142
displaying context menu 142
displaying full-screen 142
displaying type 150
exporting version 32
methods 125, 144
normal menu view 142
scaling movies to 142
forms
creating 125, 137
required elements 137
search 138
variables 139
verifying data 139
frame actions
assigning 35
assigning to keyframes 35
in conflicting layers 147
placement 35
frames, assigning actions to 35
fscommand action 125
commands and arguments 142
communicating with Director 143
using 141
functions
calling 67
constructor 14
custom 65
defined 20
defining 65
local variables in 66
passing arguments to 66
predefined 63
returning values 67
rules 63
sample 19

G

getBounds method 111
getBytesLoaded method 111
getBytesTotal method 111
getCode method 85
getting information from remote files 126
getting mouse position 82
getURL action 126
    communicating with server-side scripts 130
    search form 138
global variables 48
globalToLocal method 111
grouping statements 39

H

handlers
    checking for XML data 129
    defined 20
hard references 51
hierarchical addresses 21
hierarchy
    inheritance 74
    movie clip 99
    parent-child movie clips 100
highlighting syntax 31
hitTest action, sample 23
hitTest method 94
    controlling movies 111
HTTP protocol 126
    communicating with server-side scripts 130
HTTP requests, permitting 128
HTTPS protocol 126

I

identifiers
    defined 20
    with values 21
if statements 17, 60
importing ActionScript 30
information, passing between movie 126
inheritance, creating 74
input text 86
input text fields, in forms 137
Insert Target Path button 108
inserting target paths 108
instance names
    assigning 59
    defined 20
    movie clips 15
    setting dynamically 57
instances
    copying 14
    defined 20
instantiaating objects 69
interactivity
    complex 80
    creating 79
    forms 137
interface elements
    custom 119
    Smart Clips 119
ISO-8859-1 character set 8

J

JavaScript
    alert statement 156
    compared to ActionScript 7
    Developer Central 7
    editing 27
    international standard 7
    sending messages to 142
    supported language 8
    with statement 102

K

key codes, getting 84
Key object 84
keyboard controls 85
keyframes, assigning frame actions 35
keypresses, capturing 84
keywords
  case-sensitive 40
  defined 20
  listed 41
  syntax color 31

L

levels 60
  absolute path 104
  hierarchy 99
  loading 115
  loading movies into 98
  naming in target path 105
linking movie clips 118
List Objects command 155
List Variables command 155
LiveConnect 144
loaded data, checking for 129
loaded movies
  controlling 108
  identifying 60
loading data, security 127
loadMovie action 126
  checking for loaded data 129
  communicating with server-side scripts 130
  levels 98
loadVariables action 126
  checking for loaded data 129
  communicating with server-side scripts 130
local variables 48
  in functions 66
  sample 48
localToGlobal method 111
logical branch 17
logical operators 54
looping, children objects 62
looping actions 61

M

Macromedia Director, communicating with 143
manipulating numbers 44
maxscroll property 86
message box, displaying 143
methods 14, 45
  accessing 69
  assigning 114
  compared to actions 112
  defined 20
  invoking 112
  object 68
  targeting multiple Timelines 113
MIME format application/
    x-www-urlformencoded 130
mouse position, getting 82
movie clips
  about 97
  attaching 118
  changing properties in Debugger 153
  changing visibility 12
  controlling 108
  data type 45
  Debugger display 150
  defining clip parameters 119
  detecting collisions 94
  displaying hierarchy 99
  displaying properties 153
  dragging 117
  duplicating 15, 117
  exchange 124
  giving instance name 59
  graphic representation 14
  hierarchical relationship 100
  inserting target path 26
  instance names 15
  listing objects 155
  removing 117
  sharing 118
Movie Explorer 146
  display 104
MovieClip object
  about 15
  controlling movies 111
MovieClip objects, using 71
moviename_DoFSCommand 142

movies
controlling in Flash Player 144
listing variables 155
loading additional 115
maintaining original size 142
passing information between 126
replacing with loaded movie 115
scaling to Flash Player 142
securing 127
testing in browser 146
unloading 115
moving clips, looping children 62
multidimensional arrays 57

N

name collisions 48
names 20
naming conventions 146
naming variables 46
navigation, controlling 79
Netscape DevEdge Online 7
Netscape plug-in 150
new operator 69
nodes 131
Normal Mode 25
calling function 64
numbers 44
converting to 32-bit integers 55
numeric operators 53

O

Object Actions panel 22
object initializer operator 69
object methods, calling 70
object properties
accessing 70
object-oriented scripting 14
objects 14
assigning actions 33
creating 69
creating custom 73
custom 72
data type 45
defined 20
predefined 68
onClipEvent(enterFrame), sample 23
onClipEvent(load), sample 23
opening, Flash 4 files 75
opening message box 143
operators
array access 57
assignment 56
associativity 52
bitwise 55
combining with values 51
comparison 53
defined 20
dot 57
equality 56
logical 54
numeric 53
string 54
Output window
List Objects command 155
List Variables command 155
options 154
using 154

P

parameters
arguments and 66
changing 26
displaying 34
entering 26
passing to functions 66
Parameters fields 26
_parent alias 106
parent-child relationships 100
passing values
by content 49
by reference 50
passwords
creating 128
Debugger 149
placeholders 19
planning scripts 13
ports, XMLSocket connection 128
predefined functions 63
listed 63
predefined objects, listed 68
preferences, editing mode 28

primitive data types 42
  Flash 4 77
printing actions 30
projectors, executing application 142
properties 14
  collections 20
  defined 21
  syntax color 31
  unchanging 42
Properties tab 153
prototype property 74

R

reference data types 42
references, permanent 51
referencing variables 48
relative target path 104
remote files, communicating with 126
remote sites, continuous connection 135
removeMovieClip action 117
removing
  loaded movies 115
  movie clips 117
reordering actions 26
repeating actions 61
reserved words 20
  listed 41
  this 23
RGB method 88
RRB Color Value List 431

S

sample movie 22
saving scripts 146
Script window, changing font 29
scripting ActionScript 12
scripts
  commenting 146
  controlling execution 18
  controlling flow 60
  debugging 148
  declaring variables 49
  execution order 15
  flow 15
  guidelines 145
  importing 30
scripts *(continued)*
  planning 13
  sample 22
  searching 30
  troubleshooting 145
  writing 37
scroll property 86
scrolling text fields 86
search fields 138
security 127
  standard HTML 128
sending information
  to remote files 126
  URL encoded format 126
  via TCP/IP socket connection 126
  XMLformat 126
server-side scripts
  languages 126
  XML format 132
set variable action, verifying data 139
setPan method 90
setTransform method 88
setVolume method 90
Shift-JIS character set 8
Show Deprecated Syntax command 32
slash syntax 39
  target paths 105
Smart Clips
  creating 119
  setting clip parameters 122
socket connections 135
  sample script 136
soft references 51
Sound object 90
sounds
  attaching 91
  balance control 93
  creating volume controls 90
special characters 43
statements
  grouping 39
  logical branches 17
  reordering 26
  setting as expressions 147
  terminating 39
status bar, Debugger 150
string operators 54

strings 43
escaping characters 43
syntax color 31
Submit button 138
swapDepths method 111
Symbol Linkage Properties dialog box 118
syntax
case-sensitivity 40
curly braces 39
dot 38
parentheses 40
rules 37
semicolon 39
slash 39
syntax errors
checking 31
highlighting 32
identifying 31
syntax highlighting 31
deprecated 32
turning on and off 31

T

target paths 104
defined 21
entering 60
expression 110
inserting 26
level names 105
specifying 59, 108
targeting
duplicateMovieClip action 102
targetPath function 110
TCP/IP connection
sending information 126
with XMLSocket object 135
terminating statements 39
terms, defined 19
Test Movie command 33, 146
testing
movies 146
scripts 146
variable values 49
testing actions 33
testing frame actions 36
test-movie mode 147
text, searching for in scripts 30
text fields 125
scrolling 86
this 23
current Timeline alias 106
Timelines
communicating between 102
controlling 111
multiple 98
parent alias 106
targeting with multiple actions 113
Toolbox list, resizing 26
troubleshooting
checklist 147
guidelines 145
listing objects 155
listing variables 155
overview 145
using the Output window 154
with trace action 156
typing variables 47

U

unloadMovie action 115
URL subdomains 127

V

values, manipulating in expressions 51
variables
absolute path 152
assigning multiple 56
changing values in Debugger 151
converting to XML 133
declaring 49
defined 21
determining type 47
hidden 139
in forms 139
loading from remote files 126
modifying in Debugger 150
naming 46
naming meaningfully 146
passing content 49
passing with Smart Clips 119
referencing value 50
removing from Watch list 152
rules 46

variables *(continued)*
scoping 48
sending to remote files 126
setting dynamically 57
testing 49
tracking values with text fields 146
using in scripts 49
verifying 139
Variables tab 150
VBScript 27
verifying entered data 139
sample script 140
volume
controls 90
sliding control 92

W

Watch list, Debugger 152
Web applications
continuous connection 135
integrating Flash with 125
with action 102
targeting multiple Timelines 113
wrapper action 9
writing scripts 37

X

xch instance name 124
XML 131
hierarchy 131
in server-side scripts 132
sample variable conversion 132
sending information via TCP/IP socket 126
sending information with XML methods 126
XML DOM 131
XML object methods 132
XMLSocket object
checking for data 129
methods 135
using 135